Compiling a Domain Specific Language for Dynamic Programming

Dissertation zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.) der Technischen Fakultät der Universität Bielefeld

vorgelegt von
Peter Steffen

Bielefeld, im Oktober 2006

Gedruckt auf alterungsbeständigem Papier nach ISO 9706.

Acknowledgments

I thank my supervisor Robert Giegerich for assigning me this interesting topic and for all his support during the work on this thesis. Thanks also go to Jens Stoye for his spontaneous willingness to be a co-referee.

Jens Reeder helped a lot by proofreading this thesis.

I thank my colleagues in the practical informatics department for the nice work atmosphere, especially Elke Möllmann, my office colleague for several years.

Marco Rüther worked hard for the development of the ADP compiler. Christian Lang and Georg Sauthoff's work on the table design problem was very inspiring and useful. I also thank Stefanie Schirmer, who worked on the frontend for the ADP compiler.

Finally, I thank my parents and Alexandra for all their support.

Contents

Foreword

Algebraic Dynamic Programming (ADP) is a method to develop algorithms of dynamic programming. It was first introduced by R. Giegerich in 1998 [13]. In the following years, a number of papers were published introducing additional techniques for the ADP methodology and describing applications developed with the help of ADP. The following articles were published with the author's participation. Some of the texts have been incorporated in this thesis, constituting (most of) chapters 1 and 2 and sections 3.5, 4.5 and 5.1.

- R. Giegerich, C. Meyer, and P. Steffen. Towards a discipline of dynamic programming. In S. Schubert, B. Reusch, and N. Jesse, editors, *Informatik bewegt*, GI-Edition - Lecture Notes in Informatics, pages 3–44. Bonner Köllen Verlag, 2002.
- R. Giegerich, C. Meyer, and P. Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, 2004.
- R. Giegerich and P. Steffen. Implementing algebraic dynamic programming in the functional and the imperative programming paradigm. In E.A. Boiten and B. Möller, editors, *Mathematics of Program Construction*, pages 1–20. Springer LNCS 2386, 2002.
- R. Giegerich and P. Steffen. Challenges in the compilation of a domain specific language for dynamic programming. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, 2006.
- J. Reeder, P. Steffen, and R. Giegerich. Effective ambiguity checking in biosequence analysis. *BMC Bioinformatics*, 6(153), 2005.
- M. Rehmsmeier, P. Steffen, M. Höchsmann, and R. Giegerich. Fast and effective prediction of microRNA/target duplexes. *RNA*, 10:1507–1517, 2004.
- P. Steffen and R. Giegerich. Versatile and declarative dynamic programming using pair algebras. *BMC Bioinformatics*, 6(224), 2005.
- P. Steffen and R. Giegerich. Table design in dynamic programming. *Information and Computation*, 204(9):1325–1345, 2006.
- P. Steffen, B. Voß, M. Rehmsmeier, J. Reeder, and R. Giegerich. RNAshapes: an integrated RNA analysis package based on abstract shapes. *Bioinformatics*, 22(4):500–503, 2006.

Chapter 1

Introduction

This chapter is taken from [19].

1.1 The role of dynamic programming in bioinformatics

In biological sequence analysis, there arise numerous combinatorial optimization problems that are solved by dynamic programming. Pattern matching in DNA or protein sequences, comparison for local or global similarity, and structure prediction from RNA sequences are frequent tasks, as well as the modeling of families of proteins and RNA structures with the widely used Hidden Markov Models (HMMs) and stochastic context free grammars (SCFG), respectively [8]. The scoring schemes associated with these optimization problems can be quite sophisticated. The thermodynamic model for RNA structure prediction, for example, has more than thousand parameters. This requires elaborate case analysis. Objective functions often ask for more than a single answer, such as the best non-overlapping pattern hits to a genome above a certain score threshold. Finally, biosequences tend to be long (from 77 characters for a tRNA, 10000 for a gene, 3000000 for a bacterial genome, to the $3 * 10^9$ nucleotides of a mammalian genome such as human or mouse). The time and space requirements for a dynamic programming algorithm are often limiting factors for the problems the biologists need to solve. These characteristics of the application domain are responsible for the fact that the development of reliable and efficient dynamic programming algorithms in bioinformatics is a recurring challenge, in sharp contrast to the simplicity suggested by the textbook examples of dynamic programming which we use to teach computer science students.

1.2 State of the art

All these optimization problems share the characteristics that the logical problem decomposition follows the decomposition of the input sequence into subwords. It has been observed early that the resulting dynamic programming recurrences strongly resemble those of a Cocke-Younger-Kasami [1] type parsing algorithm [46]. Pursuing this analogy, we have developed an algebraic style of dynamic programming (ADP) over sequential data. The search space of the optimization problem at hand is described by a "yield grammar", which is a regular tree grammar generating a tree language, and implicitly a context-free language as the set of leaf sequences of these trees. Scoring and optimization are described by an "evaluation algebra", which interprets the tree operators as functions that compute local score contributions, and hence solve larger problems when given optimal solutions of smaller ones, consistent with the general paradigm of dynamic programming . This leads to a complete specification of dynamic programming algorithms on a rather high level of abstraction.

1.3 The development of ADP as a domain specific language

ADP was introduced by R. Giegerich in 1998 [13]. The paper provided a simple ADP language, basically an ASCII notation for yield grammars and evaluation algebras, and an implementation of this language as a domain specific language (DSL) embedded in the functional programming language *Haskell* [37]. In the following years, a number of papers were published introducing additional techniques for the ADP methodology [15, 18, 16, 17, 49]. During the development of the ADP language also a number of applications were developed using ADP: An algorithm for the saturated RNA secondary structure folding problem [9], an RNA folding algorithm based on the concept of abstract shapes (RNAshapes) [20], an RNA folding algorithm for predicting secondary structures possibly containing pseudoknots (pknotsRG) [39], and a program that predicts multiple potential binding sites of miRNAs in large target RNAs (RNAhybrid) [42]. The ADP method made it possible to develop these programs in relatively short time. But it also turned out that the embedded nature of the ADP language has several shortcomings, the most important one the large time and space requirements (see also below). So it became obvious that an automated compilation of the embedded ADP language into an imperative programming language like C is needed. For this, the work on the ADP compiler was started in 2001 with the author's diploma thesis [48]. Until now, the tools RNAshapes, RNAhybrid and pknotsRG were successfully compiled with the help of the ADP compiler. An overview of the ADP compiler was given in [19].

1.4 Shortcomings of the embedded DSL

We have adopted the ADP method for the development of bioinformatics tools, and have trained a first generation of bioinformatics students in it. We clearly see the advantages of declarative specifications of dynamic programming algorithms, be it for education or for algorithm design on new problems. But it also became clear that much additional work needs to be done to bring this method to bear for every-day work in the field of bioinformatics.

Domain specific languages have a wide range of applications. See [53] for an extensive annotated bibliography. They can be implemented by a variety of techniques, such as compilation, interpretation, or embedding in a host language. The concept of *embedded* domain-specific languages was introduced by Hudak [25]. The main advantage of this approach is that the implementation of a new DSL is very easy, since the complete infrastructure of the base language can be reused. In case of ADP, the ease of implementation comes with severe disadvantages in using the language by its intended community. Our critique addresses five points, which are all associated with the implementation of ADP as an *embedded* DSL.

1. While it is a rewarding experience to simply write down grammar and algebra, and to obtain an executable program, frustration lures nearby: With any simple error, the developer is confronted with the error handling of the host language. For example, for an omitted nonterminal symbol in the right-hand side of a grammar rule, we get an error message from the *Haskell* type system. Bioinformaticians are not normally trained *Haskell* programmers, and it is as unlikely as it is unfortunate that this will change in the near future. To make things worse – to understand such messages, one not only needs to be *Haskell*-literate, but one also needs to know details of the embedding. (This would probably also hold with any other embedded solution.) All the abstraction provided by the ADP language is lost when an error is made.

2. A major advantage of ADP is that the logic of the algorithm can be designed largely without being concerned about efficiency. However, once the tree grammar is designed, the current implementation requires to add "efficiency annotations" to the grammar. The user has to indicate which nonterminal symbols are mapped to dynamic programming tables (storing intermediate results). It is also necessary to treat nonterminals that generate only words of bounded length in a special way, in order to achieve best polynomial runtime efficiency.

3. In the embedded implementation, the specification is literally interpreted as code in the host language. There are numerous opportunities for optimization that are missed, as they require processing steps at compile time. The standard optimizations of the host language compiler cannot be expected to find these opportunities.

4. Being compiled by the *Haskell* compiler, the resulting program suffers from the typical efficiency disadvantage often observed when comparing (non-strict) functional to imperative programs.

5. Finally, the use of a direct embedding in a host language entices programmers to occasionally use features of the host language that are not strictly part of the ADP language. This makes programs less transparent and less re-usable. Furthermore, useful theorems pertaining to the abstract level of ADP may no longer hold for such a patched implementation.

1.5 Overview of a new approach

The above points of critique can be alleviated by "exbedding" the ADP language from its host language and providing a compiled, stand-alone implementation. Aside from being a useful contribution for programmers in the field of bioinformatics, such a compiler project also poses some compilation challenges that are interesting in their own right.

Points 1 and 5 require to write a specific compiler front-end. It can ensure language integrity, and being aware of the semantics of yield grammars and evaluation algebras, it can generate meaningful error messages to programmers. In principle, such a front end could also run as a pre-processor with the current, embedded implementation. However, much is to be gained from a more ambitious approach.

Points 2, 3, and 4 require a compiler middle- and back-end that does non-trivial analyses of the ADP source program. It brings about the chance to completely free the ADP programmer from efficiency considerations, obviating the need to specify tabulation and special treatment of nonterminal symbols. In the embedded implementation, such an un-annotated program would have wasteful space and (in most cases) exponential runtime requirements; the compiler can take the responsibility to find the program's best possible space and time efficiency, and generate code that achieves optimal asymptotic performance.

The latter point is peculiar to our compiler project, as compilers normally optimize constant factors. Achieving optimal asymptotic efficiency must not be misunderstood as if the compiler was to solve theoretical problems of problem complexity. It means that a given (un-annotated) ADP specification can be implemented as a dynamic programming algorithm with different selections of tables and also with different iterative constructs, which can lead to a wide range of performances even in the asymptotic sense. We shall show that optimal asymptotic performance is in fact a well-defined, but non-trivial problem, where the compiler is likely to do a better job than human programmers with real-world applications.

1.6 Organization of this thesis

In the next section, we give a short review of the ADP language, mainly explained by example. In Chapter 3 we define the ADP language supported by the compiler. In Section 3.5 we give a short overview of the tasks needed to compile this language into imperative target code. Chapter 4 then describes these compilation steps in detail. Finally, Chapter 5

introduces RNAshapes, a non-trivial ADP program generated with the help of the ADP compiler. Chapter 5 also gives some benchmarks of several compiled ADP programs.

Chapter 2

Algebraic dynamic programming

This chapter is taken from [49].

2.1 Overview

We set the stage for our exposition with a condensed review of the "algebraic" approach to dynamic programming. Based on this programming style, we introduce a generic product operation of scoring schemes. This leads to a remarkable variety of applications, allowing us to achieve optimizations under multiple objective functions, alternative solutions and backtracing, holistic search space analysis, ambiguity checking, and more, without additional programming effort. We demonstrate the method on several applications for RNA secondary structure prediction.

2.2 Algebraic dynamic programming by example

For our presentation, we need to give a short review of the concepts underlying the algebraic style of dynamic programming (ADP): trees, signatures, tree grammars, and evaluation algebras. We strive to avoid formalism as far as possible, and give an exemplified introduction here, sufficient for our present concerns. See [17] for a complete presentation of the ADP method. As a running example, we use the RNA secondary structure prediction problem. We start with a simple approach resulting in an ADP variant of Nussinov's algorithm [34] and move on to a more elaborate example to permit the demonstration of our new concepts.

Nothing of the new ideas presented here is specific to the RNA folding problem. Products can be applied to all problems within the scope of algebraic dynamic programming, including pairwise problems like sequence alignment [17].

2.2.1 RNA secondary structure prediction

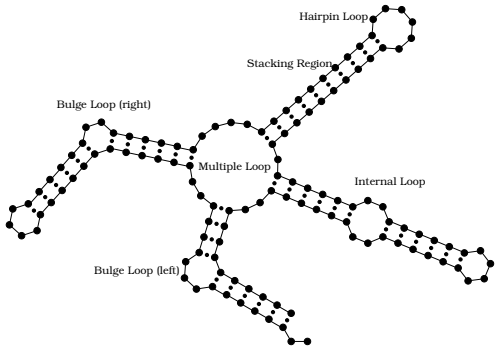


Figure 2.1: Typical elements found in RNA secondary structure.

While today the prediction of RNA 3D structure is inaccessible to computational methods, its *secondary structure*, given by the set of paired bases, can be predicted quite reliably. Figure 2.1 gives examples of typical elements found in RNA secondary structure, called stacking regions (or helices), bulge loops, internal loops, hairpin loops and multiple loops.

The first approach to structure prediction was proposed by Nussinov in 1978 and was based on the idea of maximizing the number of base pairs [34]. Today's algorithms are typically based on energy minimization.

2.2.2 ADP methodology

When designing a dynamic programming algorithm in algebraic style, we need to specify four constituents:

- Alphabet: How is the input sequence given?
- Search space: What are the elements of the search space and how can they be represented?
- Scoring: Given an element of the search space, how do we score it?
- Objective: Given a number of scores, which are the ones we are interested in?

In the following, we will work through these steps for the RNA secondary structure prediction problem.

Alphabet The input RNA sequence is a string over $\mathcal{A} = \{a, c, g, u\}$. $\mathcal{A}$ is called the alphabet and $\mathcal{A}^*$ denotes the set of sequences over $\mathcal{A}$ of arbitrary length. ε denotes the empty string. In the following, we denote the input sequence with $w \in \mathcal{A}^*$.

Search space Given the input sequence w, the search space is the set of all possible secondary structures the sequence w can form. In the ADP terminology, the elements of the search space for a given input sequence are called *candidates*. Our next task is to decide how to represent such candidates. Two possible ways are shown in Figure 2.2. The first variant is the well-known dot-bracket notation, where pairs of matching parentheses are used to denote pairing bases. The second variant, the tree representation, is the one we use in the algebraic approach.

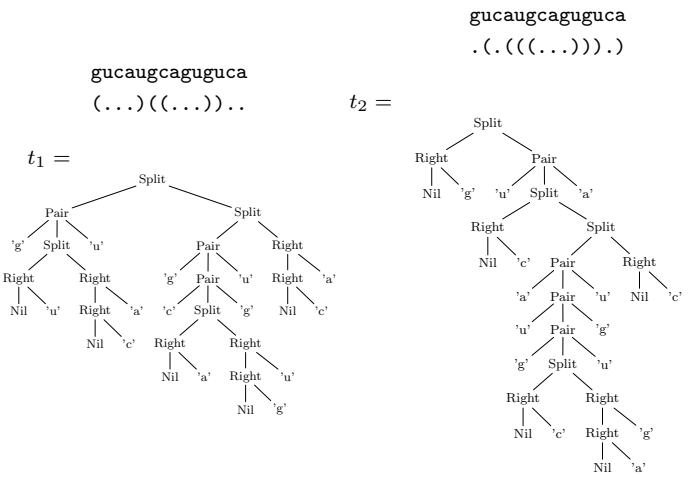


Figure 2.2: Two candidates in the search space for the best secondary structure for the sequence `gucaugcaguguca`.

Such a tree representation of candidates is quite commonly used in RNA structure analysis, but not so in other applications of dynamic programming. To appreciate the scope of the ADP method, it is important to see that such a representation exists for any application of dynamic programming [49].

In our example, the trees are constructed using four different node labels. Each label represents a different situation, which we want to distinguish in the search space and in the eventual scoring of such a candidate. A node labeled *pair* represents the paring of two bases in the input sequence. The remaining nodes *right*, *split* and *nil* represent unpaired,

branching and empty structures. It is easy to see that each tree is a suitable representation of the corresponding dot-bracket string. Also note that in each of the example trees, the original sequence can be retrieved by collecting the leaves in a counter-clockwise fashion. This is what we call the *yield* of the tree. The *yield function* y maps candidate trees back onto their corresponding sequences.

The next important concept is the notion of the *signature*. The signature describes the interface to the scoring functions needed in our algorithm. We can derive the signature for our current example by simply interpreting each of the candidate trees' node labels as a function declaration:

$$
\begin{array}{lccccccccc}
nil: & & & & \{\varepsilon\} & & & \rightarrow & Ans \\
right: & & & & Ans & \times & \mathcal{A} & \rightarrow & Ans \\
pair: & \mathcal{A} & \times & Ans & \times & \mathcal{A} & & \rightarrow & Ans \\
split: & & Ans & \times & Ans & & & \rightarrow & Ans
\end{array}
$$

The symbol Ans is the abstract result domain. In the following, Σ denotes the signature, T_Σ the set of trees over the signature Σ.

With the concepts of *yield* and *signature* we are now prepared to give a first definition of the search space:

Given an input sequence w and a signature Σ, the search space $P(w)$ is the subset of trees from T_Σ, whose yield equals w. More formally, $P(w) = \{t \in T_\Sigma | y(t) = w\}$.

This would suffice as a very rough description of the search space. In general, we want to impose more restrictions on it, for example, we want to make sure, that the operator *pair* is only used in combination with valid base pairs. For this purpose we introduce the notion of *tree grammar*. Figure 2.3 shows grammar `nussinov78`, origin of our two example trees. This grammar consists of only one nonterminal, *s*, and one production with four alternatives, one for each of the four function symbols that label the nodes. Z denotes the axiom of the grammar. The symbols *base* and *empty* are terminal symbols, representing an arbitrary base and the empty sequence. The symbol *basepairing* is a syntactic predicate that guarantees that only valid base pairs can form a *pair*-node.

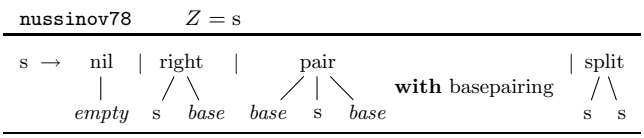


Figure 2.3: Tree grammar `nussinov78`.

Our refined definition of the search space is the following: Given a tree grammar $\mathcal{G}$ over Σ and $\mathcal{A}$ and a sequence $w \in \mathcal{A}^*$, the language described by $\mathcal{G}$ is $\mathcal{L}(\mathcal{G}) = \{t | t \in T_\Sigma,\ t$ can be derived from the axiom via the rules of $\mathcal{G}\}$. The search space spawned by w is $P_\mathcal{G}(w) = \{t \in \mathcal{L}(\mathcal{G}) | y(t) = w\}$.

From the language theoretic viewpoint, $P_{\mathcal{G}}(w)$ is the set of all parses of the sequence w for grammar $\mathcal{G}$. The method we use for constructing the search space is called *yield parsing*. See Section 2.5 for a detailed description of yield parsing.

Scoring Given an element of the search space as a tree $t \in \mathcal{L}(\mathcal{G})$, we need to score this element. In our example we are only interested in counting base pairs, so scoring is very simple: The score of a tree is the number of *pair*-nodes in t. For the two candidates of Figure 2.2 we obtain scores of 3 (t_1) and 4 (t_2). To implement this, we provide definitions for the functions that make up our signature Σ:

```
Ans_bpmax           =  ℕ
nil_bpmax(s)        =  0
right_bpmax(s,b)    =  s
pair_bpmax(a,s,b)   =  s + 1
split_bpmax(s,s')   =  s + s'
```

In mathematics, the interpretation of a signature by a concrete value set and functions operating thereon is called an algebra. Hence, scoring schemes are *algebras* in ADP. Our first example is the algebra `bpmax` for maximizing the number of base pairs. The subscript `bpmax` attached to the function names indicates, that these definitions are interpretations of the function under this algebra. In the following, we will omit these subscripts.

The flexibility of the algebraic approach lies in the fact that we don't have to stop with definition of *one* algebra. Simply define another algebra and get other results for the same search space. We will introduce a variety of algebras for our second, more elaborate example in Section 2.2.3.

Objective The tree grammar describes the search space, the algebra the scoring of solution candidates. Still missing is our optimization objective. For this purpose we add an objective function h to the algebra which chooses one or more elements from a list of candidate scores. An algebra together with an objective function forms an *evaluation algebra*. Thus algebra `bpmax` becomes:

$Ans_{\texttt{bpmax}}$ = $\mathbb{N}$

```
bpmax = (nil, right, pair, split, h) where
nil(s)          =  0
right(s,b)      =  s
pair(a,s,b)     =  s + 1
split(s,s')     =  s + s'
h([])           =  []
h([s_1, ..., s_r])  =  [ max_{1≤i≤r} s_i ]
```

A given candidate t can be evaluated in many different algebras; we use the notation $\mathcal{E}(t)$ to indicate the value obtained from t under evaluation with algebra $\mathcal{E}$.

Given that yield parsing constructs the search space for a given input, all that is left to do is to evaluate the candidates in a given algebra, and make our choice via the objective function h. For example, candidates t_1 and t_2 of Figure 2.2 are evaluated by algebra `bpmax` in the following way:

$$\begin{array}{ll} & h(\texttt{bpmax}(t_1), \texttt{bpmax}(t_2)) \\ = & [\max(3,4)] \\ = & [4] \end{array}$$

Definition 1 *(Algebraic dynamic programming)*

- *An ADP problem is specified by a signature Σ over $\mathcal{A}$, a tree grammar $\mathcal{G}$ over Σ, and a Σ-evaluation algebra $\mathcal{E}$ with objective function h.*
- *An ADP problem instance is posed by a string $w \in \mathcal{A}^*$. Its search space is the set of all its parses, $P_{\mathcal{G}}(w)$.*
- *Solving an ADP problem is computing $h\{\mathcal{E}(t) \mid t \in P_{\mathcal{G}}(w)\}$ in polynomial time and space with respect to $|w|$.*

In general, Bellman's Principle of Optimality [2] must be satisfied in order to achieve polynomial efficiency.

Definition 2 *(ADP formulation of Bellman's Principle)* *An evaluation algebra satisfies Bellman's Principle, if for each k-ary function f in Σ and all answer lists $z_1, \ldots, z_k$, the objective function h satisfies*

$$\begin{array}{l} h([f(x_1, \ldots, x_k) \mid x_1 \leftarrow z_1, \ldots, x_k \leftarrow z_k]) = \\ h([f(x_1, \ldots, x_k) \mid x_1 \leftarrow h(z_1), \ldots, x_k \leftarrow h(z_k)]) \end{array}$$

as well as

$$h(\ z \mathbin{+\!\!+} z'\ ) \quad = \quad h(\ h(z) \mathbin{+\!\!+} h(z')\ )$$

where $+\!\!+$ denotes list concatenation, and $\leftarrow$ denotes list membership.

Bellman's Principle, when satisfied, allows the following implementation: As the trees that constitute the search space are constructed by the yield parser in a bottom up fashion, rather than building them explicitly as elements of T_Σ, for each function symbol f the evaluation function $f_{\mathcal{E}}$ is called. Thus, the yield parser computes not trees, but their evaluations. To reduce their number (and thus to avoid exponential explosion) the objective function may be applied at an intermediate step where a list of alternative answers has been computed. Due to Bellman's Principle, the recursive intermediate applications of the objective function do not affect the final result.

As an example, consider the following two candidates (represented as terms) in the search space for sequence `aucg`:

$$
\begin{aligned}
t_3 = \texttt{Split}\ &\texttt{(Pair 'a' Nil 'u')} \\
&\texttt{(Right (Right Nil 'c') 'g')} \\
t_4 = \texttt{Split}\ &\texttt{(Pair 'a' Nil 'u')} \\
&\texttt{(Pair 'c' Nil 'g')}
\end{aligned}
$$

Since algebra `bpmax` satisfies Bellman's Principle, we can apply the objective function h at intermediate steps inside the evaluation of candidates t_3 and t_4:

$$
\begin{aligned}
& h(\texttt{bpmax}(t_3), \texttt{bpmax}(t_4)) \\
= \quad & h(\texttt{Split (Pair 'a' Nil 'u')} \\
& \qquad \texttt{(Right (Right Nil 'c') 'g'),} \\
& \quad \texttt{Split (Pair 'a' Nil 'u')} \\
& \qquad \texttt{(Pair 'c' Nil 'g'))} \\
= \quad & h(\texttt{Split (}h\texttt{(Pair 'a' Nil 'u',} \\
& \qquad \texttt{Pair 'a' Nil 'u'))} \\
& \qquad \texttt{(}h\texttt{(Right (Right Nil 'c') 'g',} \\
& \qquad \texttt{Pair 'c' Nil 'g')))} \\
= \quad & [\max(\max(1,1) + \max(0,1))] \\
= \quad & [\max(1+1)] \\
= \quad & [2]
\end{aligned}
$$

Given grammar and signature, the traditional dynamic programming recurrences can be derived mechanically to implement the yield parser. In the sequel, we shall use the name of a grammar as the name of the function that solves the dynamic programming problem at hand. Naturally, it takes two arguments, the evaluation algebra and the input sequence.

2.2.3 In-depth search space analysis

Note that the case analysis in the Nussinov algorithm is redundant – even the sequence `'aa'` is assigned the two trees `Right (Right Nil 'a') 'a'` and `Split (Right Nil 'a') (Right Nil 'a')`, which actually denote the same structure.

In order to study also suboptimal solutions, a non-redundant algorithm was presented in [57]. Figure 2.4 shows the grammar `wuchty98`. Here the signature has 8 function symbols, each one modeling a particular structure element, plus the list constructors (`nil, ul, cons`) to collect sequences of components in a unique way. Nonterminal symbol `strong` is used to capture structures without isolated (unstacked) base pairs, as those are known to be energetically unstable. Purging them from the search space decreases the number of candidates considerably. This grammar, because of its non-redundancy, can also be used to study combinatorics, such as the expected number of feasible structures of a sequence of length n.

This algorithm, as implemented in `RNAsubopt` [57], is widely used for structure prediction via energy minimization. The thermodynamic model is too elaborate to be presented here,

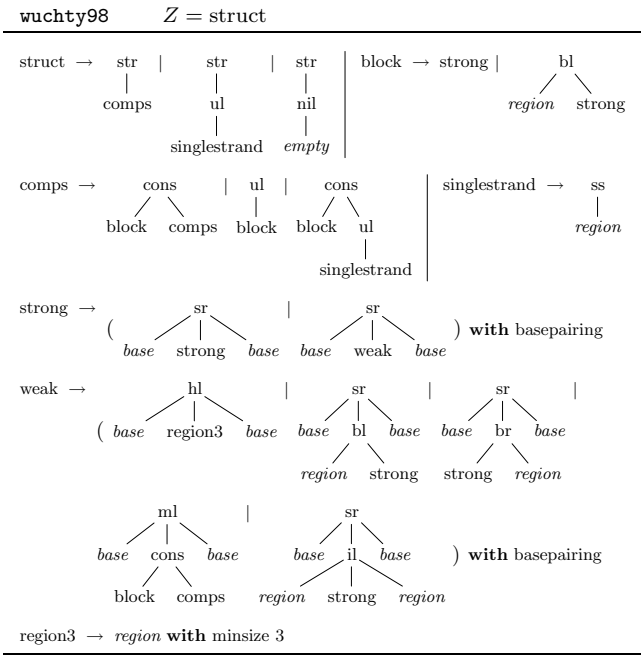


Figure 2.4: Tree grammar `wuchty98`.

$Ans_{\texttt{enum}}$ = T_Σ

```
enum = (str, ..., h) where
str(s)              =  Str s
ss((i,j))           =  Ss (i,j)
hl(a,(i,j),b)       =  Hl a (i,j) b
sr(a,s,b)           =  Sr a s b
bl((i,j),s)         =  Bl (i,j) s
br(s,(i,j))         =  Br s (i,j)
il((i,j),s,(i',j')) =  Il (i,j) s (i',j')
ml(a,s,b)           =  Ml a s b
nil((i,j))          =  Nil (i,j)
cons(s,s')          =  Cons s s'
ul(s)               =  Ul s
h([s_1,...,s_r])    =  [s_1,...,s_r]
```

$Ans_{\texttt{bpmax}}$ = $\mathbb{N}$

```
bpmax = (str, ..., h) where
str(s)              =  s
ss((i,j))           =  0
hl(a,(i,j),b)       =  1
sr(a,s,b)           =  s + 1
bl((i,j),s)         =  s
br(s,(i,j))         =  s
il((i,j),s,(i',j')) =  s
ml(a,s,b)           =  s + 1
nil((i,j))          =  0
cons(s,s')          =  s + s'
ul(s)               =  s
h([])               =  []
h([s_1,...,s_r])    =  [max_{1≤i≤r} s_i]
```

$Ans_{\texttt{pretty}}$ = dot-bracket strings

```
pretty = (str, ..., h) where
str(s)              =  s
ss((i,j))           =  dots(j-i)
hl(a,(i,j),b)       =  "("++dots(j-i)++")"
sr(a,s,b)           =  "("++s++")"
bl((i,j),s)         =  dots(j-i)++s
br(s,(i,j))         =  s++dots(j-i)
il((i,j),s,(i',j')) =  dots(j-i)++s++
                       dots(j'-i')
ml(a,s,b)           =  "("++s++")"
nil((i,j))          =  ""
cons(s,s')          =  s++s'
ul(s)               =  s
h([s_1,...,s_r])    =  [s_1,...,s_r]
```

$Ans_{\texttt{count}}$ = $\mathbb{N}$

```
count = (str, ..., h) where
str(s)              =  s
ss((i,j))           =  1
hl(a,(i,j),b)       =  1
sr(a,s,b)           =  s
bl((i,j),s)         =  s
br(s,(i,j))         =  s
il((i,j),s,(i',j')) =  s
ml(a,s,b)           =  s
nil((i,j))          =  1
cons(s,s')          =  s * s'
ul(s)               =  s
h([])               =  []
h([s_1,...,s_r])    =  [s_1 + ··· + s_r]
```

Figure 2.5: Four evaluation algebras for grammar `wuchty98`. Arguments `a` and `b` denote bases, `(i,j)` represents the input subword $w_{i+1} \dots w_j$, and `s` denotes answer values. Function `dots(r)` in algebra `pretty` yields a string of `r` dots (`'.'`)

.

and we will stick with base pair maximization as our optimization objective for the sake of this presentation. Figure 2.5 shows four evaluation algebras that we will use with grammar `wuchty98`. We illustrate their use via the following examples, where `g(a,w)` denotes the application of grammar `g` and algebra `a` to input `w`. Table 2.1 summarizes all results for an example sequence.

Application	Result
`wuchty98(enum,w)`	`[Str (Ul (Bl (0,1) (Sr 'g' (Hl 'g' (3,10) 'c') 'u'))), Str (Ul (Bl (0,2) (Sr 'g' (Hl 'g' (4,10) 'c') 'u'))), Str (Cons (Bl (0,1) (Sr 'g' (Hl 'g' (3,7) 'c') 'c')) (Ul (Ss (9,12)))), Str (Cons (Bl (0,2) (Sr 'g' (Hl 'g' (4,7) 'c') 'c')) (Ul (Ss (9,12)))), Str (Ul (Ss (0,12)))]`
`wuchty98(pretty,w)`	`[".((.......))", "..((......))", ".((....))...", "..((...))...", "............"]`
`wuchty98(bpmax,w)`	`[2]`
`wuchty98(count,w)`	`[5]`
`nussinov78(count,w)`	`[9649270]`

Table 2.1: Applications of grammars `wuchty98` and `nussinov78` with different individual algebras on input `w = cgggauaccacu`.

`wuchty98(enum,w)`: The enumeration algebra `enum` yields unevaluated terms. By convention, function symbols are capitalized in the output. Since the objective function is identity, this call enumerates all candidates in the search space spawned by `w`. This is mainly useful in program debugging, as it allows us to inspect the search space actually traversed by our program.

`wuchty98(pretty,w)`: The pretty-printing algebra `pretty` yields a dot-bracket string representation of the same structures as the above.

`wuchty98(bpmax,w)`: The base pair maximization algebra is `bpmax`, such that this call yields the maximal number of base pairs that a structure for `w` can attain. Here the objective function is maximization, and it can be easily shown to satisfy Bellman's Principle. Similarly for grammar `nussinov78`.

`wuchty98(count,w)`: The counting algebra `count` has as its objective function summation, and $\mathcal{E}_{count}(t) = 1$ for all candidates t. Hence, summing over all candidate scores gives the number of candidates. However, the evaluation functions are carefully written such that they satisfy Bellman's Principle. Thus, `[length(wuchty98(enum,w))] == wuchty98(count,w)`[1] , where the right-hand side is polynomial to compute, while the left-hand side typically is exponential due to the large number of answers returned by `wuchty98(enum,w)`.

`nussinov78(count,w)`: This computes (using an analogous version of the counting algebra not shown here) the number of structures considered by the Nussinov algorithm, which, in

[1] Technically, the result of `wuchty98(count,w)` is a singleton list, hence the [...].

contrast to the above, is much larger than the size of the search space.

These examples show analyses achieved by individual algebras. We now turn to what can be done by their combination.

2.3 The product operation on evaluation algebras

In this section we first introduce and discuss our definition of the product operation. From there, we proceed with a series of examples demonstrating its usage.

2.3.1 Definition

We define the product operation as follows:

Definition 3 *(Product operation on evaluation algebras) Let M and N be evaluation algebras over Σ. Their product M***N is an evaluation algebra over Σ and has the functions*

$$f_{M***N}((m_1,n_1)...(m_k,n_k)) = \\ (f_M(m_1,...,m_k), f_N(n_1,...,n_k)) \text{ for each } f \text{ in } \Sigma,$$

and the objective function

$$h_{M***N}([(m_1,n_1)...(m_k,n_k)]) = [(l,r)| \\ l \in L, \\ r \leftarrow h_N([r'|(l',r') \leftarrow [(m_1,n_1)...(m_k,n_k)], l' = l]), \\ \text{where } L = h_M([m_1,...,m_k]).$$

We illustrate this mechanism on the application of the product operation on algebras `bpmax` and `count`:

$Ans_{\texttt{bpmax *** count}}$ = $\mathbb{N} \times \mathbb{N}$

```
bpmax *** count = (str, ..., h) where
str((m,n))              =  (m,n)
ss((i,j))               =  (0,1)
hl(a,(i,j),b)           =  (1,1)
sr(a,(m,n),b)           =  (m + 1,n)
bl((i,j),(m,n))         =  (m,n)
br((m,n),(i,j))         =  (m,n)
il((i,j),(m,n),(i',j'))  =  (m,n)
ml(a,(m,n),b)           =  (m + 1,n)
nil((i,j))              =  (0,1)
cons((m,n),(m',n'))     =  (m + m', n * n')
ul((m,n))               =  (m,n)
h([(m_1,n_1)...(m_k,n_k)])  =
    [(l,r)|l ∈ L,
    r ← [∑[r'|(l',r') ← [(m_1,n_1)...(m_k,n_k)],l' = l]]],
    where L = [ max_{1≤i≤k} m_i]
```

Here, each function calculates a pair of two result values. The first is the result of algebra `bpmax`, the second is the result of algebra `count`. The interesting part is the objective function h. It receives the list of pairs as input, with each pair consisting of the candidate's scores for the first and for the second algebra. In the first step the objective function of algebra `bpmax` (max) is applied to the list of the first pair elements. The result is stored in L. Then, for each element[2] of L, a new intermediate list is constructed that consists of all corresponding right pair elements of the input list. This intermediate list is then applied to the objective function of the second algebra (here: summation). Finally, the result of h is constructed by combination of all elements from L with their corresponding result for the second algebra stored in r. This computes the optimal number of base pairs, together with the number of candidate structures that achieve it.

Above, $\in$ denotes set membership and hence ignores duplicates. In contrast, $\leftarrow$ denotes list membership and respects duplicates. Implementing set membership may require some extra filtering effort, but when the objective function h_M, which computes L, does not produce duplicates anyway, it comes for free.

One should not rely on intuition alone for understanding what $M{*}{*}{*}N$ actually computes. For any tree grammar $\mathcal{G}$ and product algebra $M{*}{*}{*}N$, their combined meaning is well defined by Definition 1, and the view that a complete list of all candidates is determined first, with $h_{M{*}{*}{*}N}$ applied only once in the end, is very helpful for understanding. But does $\mathcal{G}(M{*}{*}{*}N, w)$ actually do what it means? The implementation works correctly if and only if Bellman's Principle is satisfied by $M{*}{*}{*}N$, which is *not* implied when it holds for M and N individually! Hence, use of product algebras is subject to the following

Proof obligation: Prove that $M{*}{*}{*}N$ satisfies Bellman's Principle (Definition 2).

Alas, we have traded the need of writing and debugging code for a proof obligation. Fortunately, there is a theorem that covers many important cases[3]:

Theorem 1 *(1) For any algebras M and N, and answer list x, $(id_M * * * id_N)(x)$ is a permutation of x.*

(2) If h_M and h_N minimize with respect to some order relations $\leq_M$ and $\leq_N$, then $h_{M{*}{*}{*}N}$ minimizes with respect to the lexicographic ordering $(\leq_M, \leq_N)$.

Proof. (1) According to Definition 3, the elements of x are merely re-grouped according to their first component. For this to hold, it is essential that L is treated as a set. (2) follows directly from Definition 3. □

[2] In this example, L holds only one element, namely the maximum base pair score of the input list.

[3] In [49], the theorem erroneously contains also the following statement: If both M and N minimize as above and both satisfy Bellman's Principle, then so does $M{*}{*}{*}N$. This property would be nice to have, but there exist counterexamples where it does not hold.

In the above proof, *strict* monotonicity is required only if we ask for multiple optimal, or for the k best solutions rather than a single, optimal one [33].

Theorem 1 (1) justifies our peculiar treatment of the list L as a set. It states that no elements of the search space are lost or get duplicated by the combination of two algebras. Theorem 1 (2,3) say that `***` behaves as expected in the case of optimizing evaluation algebras. This is very useful, but not too surprising. A surprising variety of applications arises when `***` is used with algebras that do not do optimization, like `enum`, `count`, and `pretty`.

The proof obligation is met for all the applications studied below. A case where the proof fails is, for example, `wuchty98(count***count,w)`, which consequently delivers no meaningful result.

2.3.2 Implementing the product operation

The algebraic style of dynamic programming can be practiced within any decent programming language. It is mainly a discipline of structuring our dynamic programming algorithms, the perfect separation of problem decomposition and scoring being the main achievement. When using the ASCII notation for tree grammars proposed in [17], the grammars can be compiled into executable code. Otherwise, one has to derive the explicit recurrences and implement the corresponding yield parser. Care must be taken to keep the implementation of the recurrences independent of the result data type, such that they can be run with different algebras, including arbitrary products.

All this given, the extra effort for using product algebras is small. It amounts to implementing the defining equations for the functions of $M{*}{*}{*}N$ generically, i.e. for arbitrary evaluation algebras M and N over the common signature Σ. In a language which supports functions embedded in data structures, this is one line per evaluation function, witnessed by our implementation in Haskell (available for download). In other languages, abstract classes (Java) or templates (C++) can be used. It is essential to provide a generic product implementation. Otherwise, each new algebra combination must be hand-coded, which is not difficult to do, but tedious and error-prone, and hence necessitates debugging. A generic product, once tested, guarantees absence of errors for all combinations.

2.3.3 Efficiency discussion

Before we turn to the uses of `***`, a word on computational efficiency seems appropriate. Our approach requires to structure programs in a certain way. This induces a small (constant factor) overhead in space and time. For example, we must generically return a list of results, even with analyses that return only a single optimal value. Normally, each evaluation function is in $O(1)$, and when h returns a single answer, asymptotic efficiency is solely determined by the tree grammar [17]. This asymptotic efficiency remains unaffected when we use a product algebra. Each table entry now holds a pair of answers, each of size $O(1)$.

Things change when we employ objective functions that produce multiple results, as the size of the desired output can become exponential in n, and then it dominates the overall computational effort. For example, the optimal base pair score may be associated with a large number of co-optimal candidates, especially when the grammar is ambiguous. Thus, if using `***` makes our programs slower (asymptotically), it is not because of an intrinsic effect of the product operation, but because we decide to do more costly analyses by looking deeper into the search space.

The only exception to this rule is the situation where objective function h_M produces duplicate values, which must be filtered out, as described with Definition 3. In this case, a non-constant factor proportional to the length of answer lists is incurred.

The concrete effect of using product algebras on CPU time and space is difficult to measure, as the product algebra runs a more sophisticated analysis than either single one. For an estimation, we measure the (otherwise meaningless) use of the same algebra twice. We compute `wuchty98(bpmax,w)` and compare to `wuchty98(bpmax***bpmax,w)`. The outcome is shown in Table 2.2. For input lengths from 200 to 1600 bases, the product algebra uses 9.57% to 21.34% more space and is 18.97% to 29.46% slower than the single algebra.

	`|w|`	`wuchty98(bpmax,w)`	`wuchty98(bpmax***bpmax,w)`	%
time (sec)	200	0.58	0.69	+ 18.97
space (MB)	200	1.88	2.06	+ 9.57
time (sec)	400	4.65	6.02	+ 29.46
space (MB)	400	4.60	5.37	+ 16.74
time (sec)	800	52.04	65.54	+ 25.94
space (MB)	800	15.61	18.77	+ 20.24
time (sec)	1600	590.72	725.03	+ 22.74
space (MB)	1600	59.85	72.62	+ 21.34

Table 2.2: Measuring time and space requirements of the product operation. All results are for a C implementation of `wuchty98`, running on a 900 MHz Ultra Sparc 3 CPU under Sun Solaris 10. The space requirements were measured using a simple wrapper function for `malloc`, that counts the number of allocated bytes. Times were measured with `gnu time`.

2.3.4 Applications of product algebras

We now turn to applications of product algebras. Table 2.3 summarizes all results of the analyses discussed in the sequel, for a fixed example RNA sequence.

Application 1: Backtracing and co-optimal solutions

Often, we want not only the optimal answer value, but also a candidate which achieves the optimum. We may ask if there are several optimal candidates. If so, we may want to see them all, maybe even including some near-optimal candidates. The traditional technique

Application	Result
`wuchty98(bpmax***count,w)`	`[(2,4)]`
`wuchty98(bpmax***pretty,w)`	`[(2,".((.......))"), (2,"..((......))"), (2,".((....))..."), (2,"..((...))...")]`
`wuchty98(bpmax***enum,w)`	`[(2,Str (Ul (Bl (0,1) (Sr 'g' (Hl 'g' (3,10) 'c') 'u'))))),(2,Str (Ul (Bl (0,2) (Sr 'g' (Hl 'g' (4,10) 'c') 'u'))))),(2,Str (Cons (Bl (0,1) (Sr 'g' (Hl 'g' (3,7) 'c') 'c')) (Ul (Ss (9,12))))),(2,Str (Cons (Bl (0,2) (Sr 'g' (Hl 'g' (4,7) 'c') 'c')) (Ul (Ss (9,12)))))]`
`wuchty98(bpmax***(enum***pretty),w)`	`[(2,(Str (Ul (Bl (0,1) (Sr 'g' (Hl 'g' (3,10) 'c') 'u'))), ".((.......))")), (2,(Str (Ul (Bl (0,2) (Sr 'g' (Hl 'g' (4,10) 'c') 'u'))), "..((......))")), (2,(Str (Cons (Bl (0,1) (Sr 'g' (Hl 'g' (3,7) 'c') 'c')) (Ul (Ss (9,12)))), ".((....))...")), (2,(Str (Cons (Bl (0,2) (Sr 'g' (Hl 'g' (4,7) 'c') 'c')) (Ul (Ss (9,12)))), "..((...))...")),(2,"..((...))...")]`
`wuchty98(shape***count,w)`	`[("_[_]",2), ("_[_]_",2), ("_",1)]`
`wuchty98(bpmax(5)***shape,w)`	`[(2,"_[_]"), (2,"_[_]_"), (0,"_")]`
`wuchty98(bpmax(5)***(shape***count),w)`	`[(2,("_[_]",2)), (2,("_[_]_",2)), (0,("_",1))]`
`wuchty98(shape***bpmax,w)`	`[("_[_]",2), ("_[_]_",2), ("_",0)]`
`wuchty98(bpmax***pretty',w)`	`[(2,".((....))...")]`
`wuchty98(pretty***count,w)`	`[(".((.......))",1), ("..((......))",1), (".((....))...",1), ("..((...))...",1), (".....",1)]`

Table 2.3: Example applications of product algebras with grammar `wuchty98` on input `w = cgggauaccacu`.

is to store a table of intermediate answers and backtrace through the optimizing decisions made [21]. This backtracing can become quite intricate to program if we ask for more than one candidate. We can answer these questions without additional programming efforts using products:

`wuchty98(bpmax***count,w)` computes the optimal number of base pairs, together with the number of candidate structures that achieve it.

`wuchty98(bpmax***enum,w)` computes the optimal number of base pairs, together with all structures for `w` that achieve this maximum, in their representation as terms from T_Σ.

`wuchty98(bpmax***pretty,w)` does the same as the previous call, producing the string representation of structures.

`wuchty98(bpmax***(enum***pretty),w)` does both of the above.

To verify all these statements, apply Definition 3, or visit the ADP web site and run your own examples.

It is a nontrivial consequence of Definition 3 that the above product algebras in fact give *all* co-optimal solutions. Should only a single one be desired, we would use `enum` or `pretty` with a modified objective function h that retains only one element.

Note that our substitution of backtracing by a "forward" computation does not affect asymptotic runtime efficiency. With `bpmax***enum`, for example, the algebra stores in each table entry the optimal base pair count, together with the top node of the optimal candidate(s) and pointers to its immediate substructures, which reside in other table entries. Hence, even if there should be an exponential number of co-optimal answers, they are represented in polynomial space, because subtrees are shared. Should the user decide to have them all printed, exponential effort is incurred, just as with a backtracing implementation.

Application 2: Holistic search space analysis

Abstract shapes were recently proposed in [20] as a means to obtain a holistic view of an RNA molecule's folding space, avoiding the explicit enumeration of a large number of structures. Bypassing all relevant mathematics, let us just say that an RNA shape is an object that captures structural features, like the nesting pattern of stacking regions, but not their size. We visualize shapes by strings alike dot-bracket notation, such as `_[_[_]]`, where `_` denotes an unpaired region and `[` together with the matching `]` denotes a complete helix of arbitrary length. This is achieved by the following shape algebra:[4]

[4] The function `shMerge` appends two strings and merges adjacent characters for unpaired regions (`_`). The function `nub` eliminates duplicates from its input list.

$Ans_{\texttt{shape}}$	=	shape strings
`shape = (str, ..., h) where`		
`str(s)`	=	`s`
`ss((i,j))`	=	`"_"`
`hl(a,(i,j),b)`	=	`"[_]"`
`sr(a,s,b)`	=	`s`
`bl((i,j),s)`	=	`"_"++s`
`br(s,(i,j))`	=	`s++"_"`
`il((i,j),s,(i',j'))`	=	`"_"++s++"_"`
`ml(a,s,b)`	=	`"["++s++"]"`
`nil((i,j))`	=	`""`
`cons(s,s')`	=	`shMerge(s,s')`
`ul(s)`	=	`s`
`h(`$[s_1, \ldots, s_r]$`)`	=	`nub`$[s_1, \ldots, s_r]$

Together with a creative use of `***`, the shape algebra allows us to analyze the number of possible shapes, the size of their membership, and the (near-) optimality of members, and so on. Let `bpmax(k)` be `bpmax` with an objective function that retains the best `k` answers (without duplicates).

`wuchty98(shape***count,w)` computes all the shapes in the search space spawned by `w`, and the number of structures that map onto each shape.

`wuchty98(bpmax(k)***shape,w)` computes the best `k` base pair scores, together with their candidates' shapes.

`wuchty98(bpmax(k)***(shape***count),w)`
computes base pairs and shapes as above, plus the number of structures that achieve this number of base pairs in the given shape.

`wuchty98(shape***bpmax,w)` computes for each shape the maximum number of base pairs among all structures of this shape.

Application 3: Optimization under lexicographic orderings

Theorem 1 is useful in practice as one can test different objectives independently and then combine them in one operation. A simple case of using two orderings would be the following: Assume we have a case with a large number of co-optimal solutions. Let `pretty'` be `pretty` with $h = min$.

`wuchty98(bpmax***pretty',w)` computes among several optimal structures the one which comes alphabetically first according to its string representation.[5]

[5] Of course, there are more meaningful uses of a primary and a secondary optimization objective. For lack of space, we refrain from introducing another optimizing algebra here.

Application 4: Testing ambiguity

Dynamic programming algorithms can often be written in a simpler way if we do not care whether the same solution is considered many times during the optimization. This does not affect the overall optimum. A dynamic programming algorithm is then called redundant or ambiguous. In such a case, the computation of a list of near-optimal solutions is cumbersome, as it contains duplicates whose number often has an exponential growth pattern. Also, search space statistics become problematic – for example, the counting algebra speaks about the algorithm rather than the problem space, as it counts evaluated, but not necessarily distinct solutions. Tree grammars with a suitable probabilistic evaluation algebra implement stochastic context free grammars (SCFGs) [8]. The frequently used statistical scoring schemes, when trying to find the answer of maximal probability (the Viterbi algorithm, cf. [8]), are fooled by the presence of redundant solutions. In principle, it is clear how to control ambiguity [14]. One needs to show unambiguity of the *tree* grammar[6] in the language theoretic sense, and the existence of an injective mapping from T_Σ to a canonical model of the search space. However, the proofs involved are not trivial. Rather, one would like to implement a check for ambiguity that is applicable for any given tree grammar, but this may be difficult or even impossible, as the problem is closely related to ambiguity of context free languages, which is known to be formally undecidable.

Recently, Dowell and Eddy showed that ambiguity really matters in practice for SCFG design, and they suggest a procedure for ambiguity testing [7]. This test uses a combination of Viterbi and Inside algorithms to check whether the (putatively optimal) candidate returned by Viterbi has an alternative derivation. A more complete test is the following, and due to the use of `***`, it requires no implementation effort:

The required homomorphism from the search space to the canonical model may be coded as another evaluation algebra. In fact, if we choose the dot-bracket string representation as the canonical model, algebra `pretty` does exactly this. We can test for ambiguity by testing injectivity of `pretty` – by calling `wuchty98(pretty***count,w)` on a number of inputs `w`. If any count larger than 1 shows up in the results, we have proved ambiguity. This test is strictly stronger than the one by Dowell and Eddy, which detects ambiguity only if it occurs with the (sampled) "near-optimal" predictions. This and other test methods are studied in detail in [41].

Limitations of the product operation

The above applications demonstrate the considerable versatility of the algebra product. In particular, since a product algebra is an algebra itself, we can work with algebra triples, quadruples, and so on. All of these will be combined in the same fashion, and here we reach the limits of the product operation.

[6] Not the associated *string* grammar – it is always ambiguous, else we would not have an optimization problem.

The given definition of `***` is not the only way needed to combine two algebras. In abstract shape analysis [20], we use three algebras `mfe` (computing minimal free energy), `shape` and `pretty`. A shape representative structure is the structure of minimal free energy within the shape. Similarly to the above, `wuchty98(shape***(mfe***pretty),w)` computes the representatives of *all* shapes. However, computing *only* the k best shape representatives requires minimization within and across shapes, which neither `mfe***shape` nor `shape***mfe` can achieve. Hence, a hand-coded combination of the three algebras is necessary for this particular analysis.

2.4 Conclusion

We hope to have demonstrated that the evaluation algebra product as introduced here adds a significant amount of flexibility to dynamic programming. We have shown how ten meaningful analyses with quite diverse objectives can be obtained by using different products of a few simple algebras. The techniques we have shown here pertain not just to RNA folding problems, but to all dynamic programming algorithms that can be formulated in the algebraic style.

The benefits from using a particular coding discipline do not come for free. There is some learning effort required to adapt to a systematic approach and to abandon traditional coding habits. After that, the discipline pays back by making programmers more productive. Yet, the pay-off is hard to quantify. We therefore conclude with a subjective summary of our experience as bioinformatics toolsmiths.

After training a generation of students on the concepts presented here, we have enjoyed a boost in programming productivity. Four bioinformatics tools have been developed using this approach – `pknotsRG` [39], `RNAshapes` [20], `RNAhybrid` [42] and `RNAcast` [40]. The "largest" example is the pseudoknot folding grammar, which uses 47 nonterminal symbols and a 140-fold case distinction. The techniques described here have helped to master such complexity in several ways:

- The abstract formulation of dynamic programming recurrences in the form of grammars makes it easy to communicate and refine algorithmic ideas.
- Writing non-ambiguous grammars for optimization problems allows us to use the same algorithm for mathematical analysis of the search space.
- Ambiguity checking ensures us that such analyses are correct, that probabilistic analyses are not fooled by redundant recurrences, and that near-optimal solutions when reported do not contain duplicates.
- `enum` algebras allow for algorithm introspection – we can obtain a protocol of all solution candidates the program has seen, a quite effective program validation aid.

- Using `pretty` and `enum` algebras in products frees us from the tedious programming of backtracing routines.
- The use of the product operation allows us to create new analyses essentially with three key strokes – and a proof obligation that must be met.

This has created a good testbed for the development of new algorithmic ideas, which can immediately be put to practice.

2.5 Algebraic Dynamic Programming as a domain specific language

This section is taken from [17].

ADP has been designed as a domain specific language embedded in Haskell [37]. An algorithm written in ADP notation can be directly executed as a Haskell program. Of course, this requires that the functions of the evaluation algebra are coded in Haskell. A smooth embedding is achieved by adapting the technique of parser combinators [26], which literally turn the grammar into a parser. Hutton's technique applies to context free grammars and string parsing. We need to introduce suitable combinator definitions for yield parsing, and add tabulation.

Generally, a parser is a function that, given a subword of the input, returns a list of all its parses:

```
> type Subword  = (Int,Int)
> type Parser b = Subword -> [b]
```

2.5.1 Lexical parsers

The lexical parser `achar` recognizes any single character. Parser `astring` recognizes a (possibly empty) subword and parser `astringp` a nonempty subword. Specific characters or strings are recognized by `char` and `string`. Parser `empty` recognizes the empty subword and parser `loc` provides the current location in the input sequence.

```
> empty        :: Parser ()
> empty  (i,j) =  [() | i == j]

> achar        :: Parser Char
> achar  (i,j) =  [z!j | i+1 == j]

> astring       :: Parser Subword
> astring (i,j) =  [(i,j) | i <= j]
```

```
> astringp       :: Parser Subword
> astringp (i,j) =  [(i,j) | i < j]

> char         :: Char -> Parser Char
> char c (i,j) =  [c | i+1 == j, z!j == c]

> string       :: String -> Parser Subword
> string s (i,j) = [(i,j)| and [z!(i+k) == s!!(k-1) | k <-[1..(j-i)]]

> loc         :: Parser Int
> loc  (i,j)  =  [i | i == j]
```

The parsers `base` and `region` used in Section 2.2.3 are synonyms for the parsers `achar` and `astring`.

2.5.2 Nonterminal parsers

The nonterminal symbols in the grammar are interpreted as parsers, with the productions serving as their mutually recursive definitions. Each righthand side is an expression that combines parsers using the parser combinators.

2.5.3 Parser combinators

The operators introduced in the ADP notation are now defined as parser combinators: `|||` concatenates result lists of alternative parses, `<<<` grabs the results of subsequent parsers connected via `~~~` and successively "pipes" them into the algebra function on its left. Combinator `...` applies the objective function to a list of answers.

```
> infixr 6 |||
> (|||)          :: Parser b -> Parser b -> Parser b
> (|||) r q (i,j) = r (i,j) ++ q (i,j)

> infix  8 <<<
> (<<<)          :: (b -> c) -> Parser b -> Parser c
> (<<<) f q (i,j) =  map f (q (i,j))

> infixl 7 ~~~
> (~~~)          :: Parser (b -> c) -> Parser b -> Parser c
> (~~~) r q (i,j) =  [f y | k <- [i..j], f <- r (i,k), y <- q (k,j)]

> infix  5 ...
> (...)          :: Parser b -> ([b] -> [b]) -> Parser b
> (...) r h (i,j) = h (r (i,j))
```

Note that the operator priorities are defined such that an expression `f <<< a ~~~ b ~~~ c` is read as `((f <<< a) ~~~ b) ~~~ c`. This makes use of curried functions: the results of parser `f <<< a` are functions – i. e. calls to `f` with (only) the first argument bound.

The operational meaning of a `with`-clause can be defined by turning `with` into a combinator, this time combining a parser with a filter. Finally, the keyword `axiom` of the grammar is interpreted as a function that returns all parses for the specified nonterminal symbol and the complete input.

```
> type Filter    =  Subword -> Bool
> with           :: Parser b -> Filter -> Parser b
> with q c (i,j) =  if c (i,j) then q (i,j)  else []

> axiom          :: Parser b -> [b]
> axiom ax       =  ax (0,l)
```

When a parser is called with the enumeration algebra – i.e. the functions applied are actually tree constructors, and the objective function is the identity function – then it behaves like a proper yield parser and generates a list of trees according to Section 2.2.2. However, called with some other evaluation algebra, it computes any desired type of answer.

2.5.4 Tabulation

Adding tabulation is merely a change of data type, replacing a recursive function by a recursively defined table. Now we need a general scheme for this purpose: The function `table` records the results of a parser `p` for all subwords of an input of size `n`, and returns as its result a function that does lookup in this table. Note the essential invariance `(table n f)!(i,j) = f(i,j)`. Therefore, `table n p` is a tabulated parser, that can be used in place of parser `p`. In contrast to the latter, it does not compute the results for the same subword `(i,j)` repeatedly – here we enjoy the blessings of lazy evaluation and avoid exponential explosion.

The keyword `tabulated` is now defined as `table` bound to the global variable l, the length of the input.

```
> table     :: Int -> Parser b -> Parser b
> table n p =  (!) (array ((0,0),(n,n))
>                  [((i,j), p (i,j)) | i<- [0..n], j<- [i..n]])

> tabulated = table l
```

2.5.5 Removing futile computations

Consider the production `a = f <<< a ~~~ char`. Our definition of the `~~~` combinator splits subword (i, j) in all possible ways, including empty subwords on either side. Obviously,

`achar`, which recognizes a single character, has a fixed yield size of 1, leaving the subword $(i, j-1)$ for the yield of nonterminal symbol `a`. In this case, iteration over all splits of (i, j) into (i, k) and (k, j) is mathematically correct, but a futile effort. The only successful split can be $(i, j-1)$ and $(j-1, j)$. What is worse, since the production is left-recursive, the last split considered without need is (i, j) and (j, j), which leads to infinite recursion.

Both situations are avoided by using specializations of the `~~~` combinator that are aware of bounded yield sizes and avoid unnecessary splits. For the case of splitting off a single character, we use `-~~` and `~~-`, for a nonempty sequence on both sides we use `+~+`, while the fully general case of an arbitrary, but known yield size limit is treated by the `~~` combinator.

```
> infixl 7 ~~,-~~ ,~~-,+~+

> (-~~) q r (i,j) = [x y | i<j, x <- q (i,i+1), y <- r (i+1,j)]
> (~~-) q r (i,j) = [x y | i<j, x <- q (i,j-1), y <- r (j-1,j)]
> (+~+) q r (i,j) = [x y | k <- [(i+1)..(j-1)], x <- q (i,k), y <- r (k,j)]

> (~~) :: (Int,Int) -> (Int,Int)
>      -> Parser (b -> c) -> Parser b -> Parser c
> (~~) (l,u) (l',u') r q (i,j)
>      = [x y | k <- [max (i+l) (j-u') .. min (i+u) (j-l')],
>               x <- r (i,k), y <- q (k,j)]
```

These combinators are used in asymptotic efficiency tuning via width reduction as described in [16]. Using these special cases, our `nussinov` grammar can be written in the form

```
> nussinov78 alg = axiom s where
>   (nil,left,right,pair,split,h) = alg

>   s = tabulated (
>       nil   <<< empty                  |||
>       right <<<           s ~~- achar  |||
>       (pair <<< achar -~~ s ~~- achar)
>                 `with` basepairing     |||
>       split <<<       s +~+ s        ... h)
```

which now, as a functional program, has the appropriate efficiency of $O(n^3)$.

2.6 Tools for ADP development

In the course of this work a number of useful tools for ADP programmers were developed: Combinator optimization, automatic table design, and a program template generator. All tools are integrated in the ADP compiler, and can also be used on the ADP website (`http://bibiserv.techfak.uni-bielefeld.de/adp/`).

2.6.1 Combinator optimization

In Section 2.5.5 we have seen several variants of the `~~~` combinator. The variants are necessary to avoid needless computations. For example, in the production `s = right <<< s ~~~ achar` it is not necessary to iterate over all possible splits between `s` and `achar`. This fruitless computations can be avoided by using the combinator `~~-` instead of `~~~`. In the production `s = right <<< s ~~- achar` then only one split between `s(i,j-1)` and `achar(j-1,j)` is calculated.

For simple examples like this the choice of a suitable combinator is very easy, but for larger examples this can become quite difficult. This manual combinator optimization can lead to subscript errors that we wanted to circumvent with the use of ADP. The ADP compiler offers a combinator optimization mode that makes the manual choice of combinators superfluous. As input, the ADP compiler expects an ADP program where only the most general `~~~` combinator is used. As output the ADP compiler transforms the `~~~` combinators to their specialized variants. The implementation of this operation is described in Section 4.1.3.

2.6.2 Table design

As we have seen in Section 2.5.4, tabulation is necessary to avoid exponential explosion. For small programs with two or three nonterminals it is most often needed to tabulate every nonterminal. For larger grammars the task of tabulation is more difficult. Of course, we can simply tabulate every nonterminal of the grammar. But this is most often unnecessary and a waste of memory. In general, the table design problem is defined as follows: Find the minimal number of tables, under which the program has an asymptotically optimal runtime. This problem is NP-complete (Section 4.5). We have implemented both a brute-force approach and an approximate approach in the ADP compiler. The brute-force approach is suitable for grammars up to 40 nonterminals. For larger grammars, the approximate approach can be used.

Another task of table design is the optimization of table dimensions. In most cases quadratic tables are needed, but there exist cases where a linear table suffices. These cases are automatically identified by the ADP compiler. The compiler offers an optimization mode where a program completely without table annotations is automatically transformed to a program with optimal table design.

2.6.3 Template generator

A complete and executable ADP program consists of three types of components.

1. Problem-specific parts – Σ, $\mathcal{G}$ and $\mathcal{E}$, where $\mathcal{E}$ is the algebra that describes our desired analysis of the search space.

2. Several standard components – evaluation algebras `count` and `enum`, and other source code that depends on the underlying Σ.

3. Some prologue code that relates to the embedding of the ADP program in the host language, such as the implementation of yield parsing.

The code that needs to be written for type 2 and type 3 components is not difficult, but often amounts to more than half of the overall code. Errors introduced here may be difficult to spot. To help the user concentrate on the algorithmically interesting aspects of his work, we have automated the production of type 2 and type 3 components.

The design process with ADP starts with the specification of the signature Σ. At this point our *template generator* can be invoked to produce code for the following components:

- a polymorphic type declaration for Σ-algebras,
- the counting and the enumeration Σ-algebra,
- a data type definition for the candidate trees,
- a toy grammar that generates T_Σ,
- the necessary prologue code to import predefined functions for yield parsing.

At this point, the user has an executable ADP program, useful for minor plausibility tests like computing combinatorics of T_Σ with the counting algebra. To this program, the user may add additional algebras and grammars in arbitrary order and step by step. Declaring all algebras to be of the pre-generated type will catch a significant fraction of errors such as forgetting a function, omitting an argument, or providing one of the wrong type.

The following example shows the outcome of calling the template generator with the signature, which is reproduced as the five lines of code under the heading "`The signature:`". It also shows that the generated code is readable enough to serve as a starting point for extending the program.

```
Haskell header:

> import ADPCombinators
> import Array
> import List

The signature:

> data Alignment
>   =  Nil Char
>   |  D Char Alignment
>   |  I Alignment Char
>   |  R Char Alignment Char
```

```
>                       deriving (Eq, Show)
```

Algebra type:

```
> type Algebra alphabet answer = (
>   alphabet -> answer,
>   alphabet -> answer -> answer,
>   answer   -> alphabet -> answer,
>   alphabet -> answer -> alphabet -> answer,
>   [answer] -> [answer]
>   )
```

Enumeration algebra:

```
> enum :: Algebra Char Alignment
> enum = (nil, d, i, r, h) where
>   nil = Nil
>   d   = D
>   i   = I
>   r   = R
>   h   = id
```

Counting algebra:

```
> count :: Algebra Char Int
> count = (nil, d, i, r, h) where
>   nil a   = 1
>   d a b   = b
>   i a b   = a
>   r a b c = b
>   h []    = []
>   h xs    = [sum xs]
```

Algebra product operation:

```
> (***) :: Eq answer1 =>
>    (Algebra Char answer1) ->
>    (Algebra Char answer2) ->
>    Algebra Char (answer1,answer2)
> infix ***
> alg1 *** alg2 = (nil, d, i, r, h) where
>   (nil1, d1, i1, r1, h1) = alg1
>   (nil2, d2, i2, r2, h2) = alg2
>
>   nil a         = (nil1 a, nil2 a)
>   d a (b1,b2)   = (d1 a b1, d2 a b2)
>   i (a1,a2) b   = (i1 a1 b, i2 a2 b)
>   r a (b1,b2) c = (r1 a b1 c, r2 a b2 c)
>
>   h xs = [(x1,x2)| x1 <- nub $ h1 [ y1 | (y1,y2) <- xs],
>                    x2 <-       h2 [ y2 | (y1,y2) <- xs, y1 == x1]]
```

The yield grammar:

```
> grammar alg f = axiom alignment where
>   (nil, d, i, r, h) = alg
>
>   alignment = tabulated(
>               nil <<< achar                          |||
>               d   <<< achar -~~ alignment            |||
>               i   <<< alignment ~~- achar            |||
>               r   <<< achar -~~ alignment ~~- achar ... h)

Bind input:

>   z         = mk f
>   (_,n)     = bounds z
>   achar     = achar' z
>   axiom     = axiom' n
>   tabulated = table n
```

Chapter 3

Language definition

ADP is a flexible approach to describe dynamic programming algorithms. Particularly, in the embedded approach (Section 2.5) it is tempting to add new language constructs to an algorithm or even to program parts simply in Haskell. For the compilation of ADP programs this situation is difficult as long as the ADP language is not clearly defined. In this chapter we will therefore define the ADP language supported by the compiler. In Section 3.5 we give an overview of the tasks needed to compile this language into imperative target code.

3.1 Syntax

We describe the abstract syntax of the ADP language by the following abstract data types. A grammar has a name, an axiom, and a set of productions:

```
data Grammar = Grammar Ident Axiom [Production]
type Ident = String
type Axiom = String
```

A production has a name, it can be tabulated or not, and has a corresponding definition (*Unit*):

```
data Production = Production Ident Tabulation Unit
type Tabulation = Bool

data Unit = Terminal Ident
          | Nonterminal Ident
          | Unit :~~~ Unit
```

```
    | (Unit, YSize) :~~ (Unit, YSize)
    | Ident :<<< Unit
    | Unit `With` Ident
    | Unit ||| Unit
    | Unit ... Ident

type YSize = (YSNum, YSNum)
data YSNum = YSNum Int
           | Infinite
```

The following functions map the abstract syntax defined by the data types *Grammar*, *Production* and *Unit* to its concrete syntax for the embedded domain specific language:

```
ppGrammar (Grammar ident axiom ps) = ident ++ " = axiom " ++ axiom ++
                                     " where\n" ++ concatMap ppProduction ps

ppProduction (Production ident tab unit) = ident ++ " = " ++ ppTab tab ++ ppUnit unit
  where ppTab True  = "tabulated"
        ppTab False = ""

ppUnit (Terminal ident) = ident
ppUnit (Nonterminal ident) = ident
ppUnit (p :~~~ q) = ppUnit p ++ " ~~~ " ++ ppUnit q
ppUnit ((p, ys_p) :~~ (q, ys_q)) = ppUnit p ++ " ~~ " ++ ppUnit q
ppUnit (f :<<< p) = f ++ " <<< " ++ ppUnit p
ppUnit (p `With` f) = ppUnit p ++ " `with` " ++ f
ppUnit (p ||| q) = ppUnit p ++ " ||| " ++ ppUnit q
ppUnit (p ... h) = ppUnit p ++ " ... " ++ h
```

3.2 Terminal parsers

The ADP language offers a set of terminal parsers suitable for sequential input. This set contains the following parsers:

- *empty.* the empty word: $\mathcal{L}(\text{empty}) = \{\varepsilon\}$.
- *achar.* an arbitrary character: $\mathcal{L}(\text{achar}) = \mathcal{A}$.
- *astring.* an arbitrary sequence: $\mathcal{L}(\text{astring}) = \mathcal{A}^*$.
- *astringp.* an arbitrary non-empty sequence: $\mathcal{L}(\text{astringp}) = \mathcal{A}^+$.

- *char c.* a concrete character: $\mathcal{L}(\text{char c}) = \{c\},\ c \in \mathcal{A}$.
- *string s.* a concrete sequence: $\mathcal{L}(\text{string s}) = \{s\},\ s \in \mathcal{A}^*$.
- *loc*: the current position in the input: $\mathcal{L}(\text{loc}) = \mathbb{N}$.

3.3 Filter

To require results of certain lengths, the ADP language offers the following filters: *minsize* with $(minsize\ l)(x) = |x| \geq l$, *maxsize* with $(maxsize\ u)(x) = |x| \leq u$ and *size* with $(size\ l\ u)(x) = l \leq |x| \leq u$.

As filter for checking parts of the input, the filter *pairing* with $(pairing\ f)(x_1...x_n) = f(x_1, x_n)$ can be used. Furthermore it is possible to use arbitrary other filters, that need to be implemented in the target language by hand.

3.4 Algebra functions

As algebra functions, arbitrary functions can be used. The compiler then tries to compile these functions directly into the target language. For example, simple arithmetic expressions can be automatically processed. Functions unknown to the compiler are compiled to external function calls in the target language. As evaluation functions, the functions `maximum`, `minimum` and `sum` are directly supported. In general, the algebra functions can have the following form:

```
data Exp = ExpFA Ident [Exp]      -- function application
         | ExpVar String
         | ExpNum Int
         | ExpChar Char
         | ExpString String
         | ExpTupel [Exp]
         | ExpIF Exp Exp Exp
         | ExpBinOp Exp BinOp Exp

data BinOp = Add | Min | Mul | Div | Pow
           | Lt | Gt | Leq | Geq | Eq | Neq
           | And | Or | Append | Cons
```

3.5 Compilation challenges

This section is taken from [19].

In [17], we gave translation rules for the generation of matrix recurrences from tree grammars. This scheme serves as a translational semantics for ADP, but produces no useful target code.

In the following, we shall report on some challenges in the project of migrating the domain specific ADP language from its host language to a directly compiled implementation. The solutions to these challenges are described in detail in Chapter 4. As a running example we use the following program for a local alignment with affine gap costs:

```
sw_alignments z = axiom skipR where

  skipR     = skip_right <<< skipR ~~~ achar        |||
              skipL                                 ... h

  skipL     = skip_left  <<< achar ~~~ skipL        |||
              alignment                             ... h

  alignment = r   <<< achar ~~~ alignment ~~~ achar |||
              d   <<< achar ~~~ xDel                |||
              i   <<<           xIns      ~~~ achar |||
              nil <<< astring                       ... h

  xDel      = alignment                             |||
              dx <<< achar ~~~ xDel                 ... h

  xIns      = alignment                             |||
              ix <<< xIns ~~~ achar                 ... h
```

The central part of this grammar is production *alignment*. Each edit operation is expressed as one alternative of this production. It can either be a replacement of two characters, a deletion of a character from the first sequence, a deletion of a character from the second sequence (i.e. an insertion in the first), or an arbitrary sequence in the middle part, which terminates the alignment.

Based on the observation that for processing of biological sequence data it is often more realistic to assume mutations of longer sequence parts instead of only a single molecule, the local similarity algorithm uses a model of *affine gap costs*. For this purpose, it distinguishes between the first deletion or insertion and subsequent ones. This is modeled by the two nonterminals *xDel* and *xIns*. The two additional nonterminals, *skipR* and *skipL*, are used to express that an arbitrary number of characters can be skipped before the beginning of the local alignment.

The following shows algebra *affine* in ADP notation:

```
affine :: SmithWaterman_Algebra Char Int
affine = (nil, d, i, r, dx, ix,
          skip_right, skip_left, h) where
   nil _          = 0
   d x s          = s - 16
   i s y          = s - 16
   r a s b        = if a==b then s+4 else s-3
   dx x s         = s - 1
   ix s y         = s - 1
   skip_right a b = a
   skip_left  a b = b
   h          l   = [maximum l]
```

We score similar characters with a positive score (4) and assign penalties for mismatches (-3), insertions and deletions (-16 each). An extension of an insertion or deletion receives a penalty of -1.

For convenience, we begin with the matrix recurrences for the alignment example, derived following the translation scheme.

3.5.1 Recurrence derivation

The traditional way to describe DP algorithms is in the form of matrix recurrences, describing the calculation rules for the corresponding DP matrix entries. In case of a DP algorithm with more than a single DP matrix, these recurrences are also used to express the dependencies between table entries. The translation phase consist of two steps. In the first step, only the grammar is transformed, including symbolic representations of the algebra function calls. These general recurrences then serve as a template for specialization with the desired algebras.

The following shows the general matrix recurrence for production *alignment*:

$$
\begin{array}{l}
\text{for } j = 0 \text{ to } n \text{ do} \\
\quad \text{for } i = j \text{ to } 0 \text{ do} \\
\qquad alignment_{i,j} = h(\\
\qquad\quad [nil((i,j))|j-i \geq 0] \mathbin{++} \\
\qquad\quad [r(z_{i+1}, p_2, z_j)|j-i \geq 2, p_2 \in alignment_{i+1,j-1}] \mathbin{++} \\
\qquad\quad [d(z_{i+1}, p_2)|j-i \geq 1, p_2 \in xDel_{i+1,j}] \mathbin{++} \\
\qquad\quad [i(p_1, z_j)|j-i \geq 1, p_1 \in xIns_{i,j-1}]) \\
\qquad xDel_{i,j} = \ldots \\
\qquad xIns_{i,j} = \ldots
\end{array}
$$

The outer for-loops guarantee that calculations are performed with increasing subword length, such that optimal solutions can be computed from optimal solutions of subproblems.

Specialized for algebra *affine*, we obtain the following recurrence:

$$
\begin{array}{l}
alignment_{i,j} = \max(\\
\quad [0|j-i \geq 0] \mathbin{++} \\
\quad [\text{if } z_{i+1} == z_j \text{ then } alignment_{i+1,j-1} + 4 \\
\qquad \text{else } alignment_{i+1,j-1} - 3|j-i \geq 2] \mathbin{++} \\
\quad [xDel_{i+1,j} - 16|j-i \geq 1] \mathbin{++} \\
\quad [xIns_{i,j-1} - 16|j-i \geq 1])
\end{array}
$$

The recurrence derivation is described in detail in Section 4.2.

Apart from the recurrences, the calculation order has to be derived. For example, the table entry $alignment_{i,j}$ depends on $alignment_{i+1,j-1}$, $xDel_{i+1,j}$ and $xIns_{i,j-1}$. We have to guarantee that these entries are already available when $alignment_{i,j}$ is calculated. This is no problem in a data-driven language like *Haskell*, since all computations are made on demand. In the compiled, imperative implementation, these dependencies require an additional analysis. For the case above, the correct calculation order is already guaranteed by the two outer for-loops. Things become more difficult, when dependencies exists, where a calculation depends on a result of another table entry for the same subword. Here we have to make sure that the calculation is ordered in a way, such that all results are calculated before they are used. This problem can be solved by dependency analysis of the generated recurrences, together with a topological sort. For our running example, the order of calculation derived is *alignment*, *xDel*, *xIns*, *skipL*, and *skipR*. The dependency analysis is described in detail in Section 4.3.

These recurrences can be implemented straightforward in an imperative target language, using an array for each recurrence and a list data structure for the calculation. Compared to the embedding in *Haskell*, this step already achieves a significant speedup in constant factors. In the following we first report on optimizations that even affect the asymptotic runtime behavior. Then we show some optimizations that further improve constant factors.

3.5.2 Table design for optimal asymptotic efficiency

Our first optimizations address the memory requirements for the dynamic programming tables. The main question here is which results need to be stored in tables, and which results do not need to be stored. We can divide this question into three subproblems:

1. Determine the nonterminals whose results need to be stored,
2. determine the table dimensions,
3. and determine the table sizes.

Challenge 1: Table design The first challenge is to decide which nonterminals need to be stored in DP tables. The easiest way to handle this situation would be to introduce a DP table for *every* nonterminal symbol of the tree grammar. This approach is of course very luxurious. For example, the nonterminals *skipR* and *skipL* in the Smith-Waterman grammar could also be implemented as recursive functions without worsening the asymptotic runtime of the algorithm. The other extreme – no DP tables at all – leads of course to an exponential runtime of the algorithm. The challenge here is to find the minimal number of tables, such that the algorithm can be implemented with optimal asymptotic runtime.

Solution We have shown that this problem is NP complete (Section 4.5). Being no computational problem for such small grammars like our example here, it is a serious obstacle when dealing with larger grammars. Realistic applications use up to 50 nonterminal symbols and 150 rule alternatives. We have implemented several strategies to deal with this situation. These include user-annotations, preprocessing phases, a brute force approach for small grammars, and an approximate approach for larger grammars. The table design problem is studied in detail in Section 4.5.

Challenge 2: Table dimension Given that the table design analysis determined that a certain nonterminal needs to be tabulated, the next task is to decide on the required dimension for this table. For example, nonterminal *skipR* could be tabulated in a one-dimensional table, since only a linear number of results are needed during calculation. On the other hand, nonterminal *alignment* needs a two-dimensional tabulation.

Solution We have implemented a simple usage-analysis, where for each nonterminal the set of index bounds is derived. With this information the compiler can decide whether a nonterminal has to be tabulated as a one- or as a two-dimensional table.

Challenge 3: Table sizes Yield size analysis determines the valid input lengths for every nonterminal of the grammar (Section 4.1). For example, consider a nonterminal with a yield size of $(2, \infty)$. This means, that this nonterminal can by no means have any results for subwords shorter than length 2, and therefore no table space needs to be reserved for these results. The effect of this analysis is more impressive for nonterminals that have a limited *upper* yield size. For example, a nonterminal with yield size $(0, 30)$ can be tabulated in a table of dimension $n \times 30$ rather than $n \times n$.

Solution We have implemented yield size analysis, and incorporate this information during the code generation phase for the table memory allocation.

3.5.3 Optimization in inner loops by change of data type

The matrix recurrences can be directly translated in executable C code, adopting the list-based implementation of the *Haskell* embedding.

Challenge 4: List elimination In cases where the objective function delivers only a single result, we can further speedup this calculation by a change of data type to atomic result values.

Solution We begin with example target code for production *alignment*:

```
static void calc_alignment(int i, int j)
{
  int v1, v2, v3, v4, v5, v6, v7;

  if ((j-i) >= 0) {
    /* v1 = nil <<< astring */
    v1 = 0;
    /*  v2 = r <<< achar ~~~ p alignment ~~~ achar */
    if ((j-i) >= 2) {
      v2 = (z[i+1] == z[j]) ?
        tbl_alignment(i+1, j-1) + 4 :
        tbl_alignment(i+1, j-1) - 3;
    } else v2 = INTMIN;
    /* v3 = d <<< achar ~~~ p xDel */
    if ((j-i) >= 1) {
```

```
      v3 = tbl_xDel(i+1, j) - 16;
    } else v2 = INTMIN;
    /* v4 = i <<< p xIns ~~~ achar */
    if ((j-i) >= 1) {
      v4 = tbl_xIns(i, j-1) - 16;
    } else v2 = INTMIN;

    /*  h xs = [maximum xs] */
    v5 = v3 > v4 ? v3 : v4;
    v6 = v2 > v5 ? v2 : v5;
    v7 = v1 > v6 ? v1 : v6;

    tbl_alignment(i, j) = v7;
  }
}
```

Since each recurrence delivers only a single result in this case (the maximum value), the target code can completely get by with atomic integer result values. In other cases, e.g. when the objective function yields more than a single result, linked lists are used for the implementation.

The basic target code generation is completed by additional code for memory handling and the main loop calling the calculation procedures in the order derived by dependency analysis.

The code generation phase is studied in detail in Section 4.4.

3.5.4 Backtracing and suboptimal candidates

Challenge 5: Backtracing In most applications of dynamic programming, one is not only interested in the optimal result of the algorithm, but also in the way *how* this result was calculated. For this, the ADP language provides the operator `***`, which allows to combine algebras in a flexible manner [49]. For example, the combination `affine *** prettyprint` generates a new algebra that calculates the best score for algebra `affine`, together with the corresponding candidates generated by an algebra `prettyprint`. A drawback of this approach is its unsatisfying efficiency. For each of the $O(n^2)$ table entries, additional data structures have to be constructed for the candidates' representations. In the *Haskell* embedding, the additional effort is low, due the lazy nature of the language. In the C implementation, we can not easily simulate this laziness. We therefore implement a backtracing approach:

Solution Candidate generation is performed in two phases. In the first phase, the DP tables are filled with atomic results in the way described above. After the calculation is completely finished and each table entry is filled with its corresponding result value, the way of the calculation is reconstructed by a backtrace through the DP tables [21].

We have implemented methods for automatic creation of program code that handles this backtracing procedure. This consists of several additional program parts, and we again show only an example:

```
static struct str_Signature *back_alignment(int i, int j)
{
   struct str1 v1, v2, v3, v4, v5, v6, v7;
   /* v1 = nil <<< astring */
   if ((j-i) >= 0) {
      v1.alg_affine = 0; v1.alg_enum = new_Nil(i, j);
   }
   else { v1.alg_affine = INTMIN; v1.alg_enum = NULL; }
   /* v2 = r <<< achar ~~~ p alignment ~~~ achar */
   if ((j-i) >= 2) {
      v2.alg_affine = (z[i+1] == z[j]) ?
        tbl_alignment(i+1, j-1) + 4 :
        tbl_alignment(i+1, j-1) - 3;
      v2.alg_enum =
        new_R(z[i+1], back_alignment, i+1, j-1, z[j]);
   }
   else { v2.alg_affine = INTMIN; v2.alg_enum = NULL; }
   /* v3 = d <<< achar ~~~ p xDel */
   if ((j-i) >= 1) {
      v3.alg_affine = tbl_xDel(i+1, j) - 16;
      v3.alg_enum = new_D(z[i+1], back_xDel, i+1, j);
   }
   else { v3.alg_affine = INTMIN; v3.alg_enum = NULL; }
   /* v4 = i <<< p xIns ~~~ achar */
   if ((j-i) >= 1) {
      v4.alg_affine = tbl_xIns(i, j-1) - 16;
      v4.alg_enum = new_I(back_xIns, i, j-1, z[j]);
   }
   else { v4.alg_affine = INTMIN; v4.alg_enum = NULL;  }
   v5 = v3.alg_affine > v4.alg_affine ? v3 : v4;
   v6 = v2.alg_affine > v5.alg_affine ? v2 : v5;
   v7 = v1.alg_affine > v6.alg_affine ? v1 : v6;
   return(build_str_Signature(v7.alg_enum));
}
```

The generated code has the same structure as the code for calculation of atomic results. For each value, we also store additional information to control the backtracing. For example, the score for the replacement operation (`v2`) depends on table entry $alignment_{i+1,j-1}$ and the input elements z_{i+1} and z_j. The function `new_R` creates a data entry that stores both input elements and a pointer to the backtrace function `back_alignment` together with the corresponding indices $i+1$ and $j-1$. This procedure is repeated for the other parts of the

recurrence, and the data entry with the maximum score is stored in `v7`. Finally, function `build_str_Signature` traverses the generated data structure, logs the path of the current backtrace, and uses the stored function pointer to trigger the next backtrace step. This is repeated until a terminating rule is reached (here: *nil*).

Challenge 6: Suboptimal candidates Another chance for optimization lies in the handling of suboptimal candidates. For example, in RNA secondary structure prediction one is often not only interested in the best solution, but also in suboptimal solutions that occur in a certain range below the optimal one. In ADP, this can be simply implemented by an objective function that delivers a list of candidate scores instead of a single score.

Solution In the C implementation we again take a different approach. In order to avoid list-handling as far as possible, we postpone the calculations related to suboptimality to the backtrace phase. It follows the approach described by Wuchty et al. [57]. We have added an option to the code generation phase such that code implementing this suboptimal backtrace is generated automatically for any given ADP algorithm.

Chapter 4

Compiling ADP

This chapter describes in detail the steps needed to compile an ADP program into an imperative target program. Section 4.1 describes the yield size analysis, an elementary analysis needed for most subsequent compilation tasks. The recurrence derivation is studied in Section 4.2. Finding a suitable calculation order for these recurrences is the task of the dependency analysis. This is described in Section 4.3. The code generation phase then follows in Section 4.4. Finding good and optimal table configurations is the task of the table design phase. This phase is described in Section 4.5. Finally, the interface creation for the generated ADP programs is studied in Section 4.6.

4.1 Yield size analysis

4.1.1 Motivation

In this section we develop a method to calculate the yield sizes for an ADP program. The results of this analysis can be used in several phases of the compilation:

- When developing *embedded* ADP programs, we can abandon the specialized Next-Combinators (Section 4.1.3). For larger programs this is relatively complex and can easily lead to subtle errors. With the help of yield size analysis it is possible to automatically annotate the source program with the optimal set of combinators. The ADP developer can then fully concentrate on developing the recurrences without the need to think about specialized combinators. Moreover, we can check user-annotated programs for correctness.
- The knowledge about yield sizes allows to derive the matrix recurrences in a very elegant way (Section 4.2). At the same time, it supports finding optimal boundaries

for the generated loops in the target language.

- While generating the target program, the calculation order of the recurrences must be derived. It must be guaranteed that all values used are already calculated. The results of yield size analysis can also be used for this dependency analysis (Section 4.3).

4.1.2 Yield size analysis

The main idea of the analysis is that the yield sizes of terminal parsers are known. For example, the parser *astringp* yields at least one element, and the parser *achar* yields exactly one element. We can then easily calculate the yield sizes by structural recursion on the source grammar.

Let $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$ and $\overleftrightarrow{y}_p = (l_p,\ u_p) \in (\mathbb{N}^\infty, \mathbb{N}^\infty)$ the yield size of a parser p. For the parsers introduced in Section 3.2, we define the yield sizes as follows:

$$\begin{aligned}
\overleftrightarrow{y}_{empty} &= (0,0)\\
\overleftrightarrow{y}_{achar} &= (1,1)\\
\overleftrightarrow{y}_{astring} &= (0,\infty)\\
\overleftrightarrow{y}_{astringp} &= (1,\infty)\\
\overleftrightarrow{y}_{char\,c} &= (1,1)\\
\overleftrightarrow{y}_{string\,s} &= (|s|,|s|)\\
\overleftrightarrow{y}_{loc} &= (0,0)
\end{aligned}$$

In the following, we will demonstrate the analysis step-by-step with some examples. Let p be a production with $p = achar\ \sim\!\sim\!\sim\ astringp$. Intuitively, we can say that the yield size $\overleftrightarrow{y}_p$ can be calculated by

$$\begin{aligned}
\overleftrightarrow{y}_p &= (l_{achar} + l_{astringp},\ u_{achar} + u_{astringp})\\
&= (1+1,\ 1+\infty) = (2,\ \infty)
\end{aligned}$$

With this example, we define operator $+_y$, that sums up the yield sizes of two parsers in the following way:

$$\overleftrightarrow{y}_p\ +_y\ \overleftrightarrow{y}_q\ =\ (l_p,\ u_p)\ +_y\ (l_q,\ u_q)\ =\ (l_p + l_q,\ u_p + u_q) \tag{4.1}$$

We can now describe our example calculation by $\overleftrightarrow{y}_p\ =\ \overleftrightarrow{y}_{achar}\ +_y\ \overleftrightarrow{y}_{astringp}$. Accordingly we define operator $\cup_y$ by

$$\overleftrightarrow{y}_p\ \cup_y\ \overleftrightarrow{y}_q\ =\ (l_p,\ u_p)\ \cup_y\ (l_q,\ u_q)\ =\ (min\ l_p\ l_q,\ max\ u_p\ u_q), \tag{4.2}$$

for yield size calculation of two alternative parsers. Obviously, for $p\ =\ achar\ |||\ astringp$ the following must apply:

$$\begin{array}{lcccccl} \overleftrightarrow{y}_p & = & \overleftrightarrow{y}_{achar} & \cup_y & \overleftrightarrow{y}_{astringp} & & \\ & = & (1,\ 1) & \cup_y & (1,\ \infty) & = & (1,\ \infty) \end{array}$$

The combination of the two operators is shown in the following example:

$$\begin{array}{lll} p & = & \mathit{achar} \sim\!\sim\!\sim \mathit{achar} \qquad\qquad\qquad\qquad\qquad |||\\ & & \mathit{string}\ \text{,,hello"} \sim\!\sim\!\sim \mathit{string}\ \text{,,world"} \end{array}$$

$$\begin{array}{lll} \overleftrightarrow{y}_p = & ((1,\ 1)\ +_y\ (1,\ 1)) \cup_y & \\ & ((5,\ 5)\ +_y\ (5,\ 5)) & \\ = & (2,\ 2)\ \cup_y\ (10,\ 10) = (2,\ 10) & \end{array}$$

For dealing with filters we define another operator:

$$\overleftrightarrow{y}_p\ \cap_y\ \overset{\cap}{y}_f\ =\ (l_p,\ u_p)\ \cap_y\ (l_f,\ u_f)\ =\ (max\ l_p\ l_f,\ min\ u_p\ u_f) \tag{4.3}$$

$\overset{\cap}{y}_f \in (\mathbb{N}^\infty, \mathbb{N}^\infty)$ is called the *yield size constraint*. Another example shall demonstrate this:

$$\begin{array}{lcccccl} p & = & \mathit{astringp} & \textbf{with} & (\mathit{size}\ 4\ 8) & & \\ \overleftrightarrow{y}_p & = & (1,\ \infty) & \cap_y & (4,\ 8) & = & (4,\ 8) \end{array}$$

We can now describe the yield size analysis for a production p with $p\ =\ u_p$ completely by structural recursion:

$$\begin{array}{lll} \overleftrightarrow{y}_p = ysa_u\ u_p \quad \text{with} & & \\ \quad ysa_u\ (\mathit{Terminal}\ t) & = & \overleftrightarrow{y}_t \\ \quad ysa_u\ (\mathit{Nonterminal}\ nt) & = & \overleftrightarrow{y}_{nt} \\ \quad ysa_u\ (p :\!\sim\!\sim\!\sim\ q) & = & ysa_u\ p\ +_y\ ysa_u\ q \\ \quad ysa_u\ ((p,\ \overset{\cap}{y}_p) :\!\sim\!\sim (q,\ \overset{\cap}{y}_q)) & = & (ysa_u\ p\ \cap_y\ \overset{\cap}{y}_p)\ +_y\ (ysa_u\ q\ \cap_y\ \overset{\cap}{y}_q) \\ \quad ysa_u\ (c :\!<\!<\!<\ p) & = & ysa_u\ p \\ \quad ysa_u\ (p\ \text{`With'}\ f) & = & ysa_u\ p\ \cap_y\ \overset{\cap}{y}_f \\ \quad ysa_u\ (p\ :|||\ q) & = & ysa_u\ p\ \cup_y\ ysa_u\ q \\ \quad ysa_u\ (p\ :\ldots\ h) & = & ysa_u\ p \end{array}$$

Fixpoint iteration

In the above definitions we assumed a given yield size $\overleftrightarrow{y}_{nt}$ for a nonterminal nt. Since productions can be defined mutually recursive, we can not imply this here. Hence, we initialize the yield sizes of all productions with $init = \{\overleftrightarrow{y}_{nt} := (\infty,\ 0) \mid (nt = u) \in P\}$ and do the analysis with a fixpoint iteration approach. Let P be the set of all productions and $Y = \{\overleftrightarrow{y}_{nt} := (l_{nt}, u_{nt}) \mid (nt = u) \in P\}$ the set of the corresponding yield sizes. Then the fixpoint iteration is defined as follows:

$$
\begin{aligned}
Y = fix\ \ init \quad & \text{with} \\
fix\ \ current = & \begin{cases} current & \text{for } current \equiv new \\ fix\ new & \text{otherwise} \end{cases} \\
new = \{ \overleftrightarrow{y}_{nt} := ysa_u\ u \mid (nt\ =\ u) \in P \}
\end{aligned}
$$

In other words, the fixpoint iteration is repeated until two succeeding yield size analyses have the same results. Here the question about termination arises. The function ysa_u is monotonously falling in the first component l and monotonously rising in the second component u. l always converges, since $\mathbb{N}^\infty$ is downward limited by 0. u does not necessarily converge, but we can easily show that $u_p = \infty$ must apply, when u_p still rises after $|P|$ iterations:

Let $P = \{p_1, ..., p_n\}, n = |P|$. Let $p_i \to p_j$ be an appearance of the nonterminal p_j in production p_i. Accordingly, let $p_i \to^* p_j = p_i \to p_j \vee p_i \to p_k \to^* p_j$ be a dependence of production p_i on p_j. If the productions depend circularly on each other, this cycle can have at most length $|P|$. In this case the production possesses an infinite derivation, such that u_p can be set to ∞, if it is still rising after $|P|$ iterations.

Example:

	$test$	=	$achar$ $empty$	$\sim\sim\sim$	$test$	$\sim\sim\sim$	$achar$	$\mid\mid\mid$
1.	$\overleftrightarrow{y}_{test}$	=	$(1,\ 1)$ $(0,\ 0)$	$+_y$	$(\infty,\ 0)$	$+_y$	$(1,\ 1)$	$\cup_y$
		=	$(0,\ 2)$					
2.	$\overleftrightarrow{y}_{test}$	=	$(1,\ 1)$ $(0,\ 0)$	$+_y$	$(0,\ 2)$	$+_y$	$(1,\ 1)$	$\cup_y$
		=	$(0,\ 4)$					
		...						
		=	$(0,\ \infty)$					

Obviously, $\overleftrightarrow{y}_{test} = (0,\ \infty)$ is correct.

At the same time, we can identify useless productions with this analysis. A production p is useless, if $l_p = \infty$ when reaching the fixpoint. p can not posses a finite derivation here. Removing the terminal parser *empty* from the above example demonstrates this:

$$
\begin{array}{llcccccc}
 & test & = & achar & \sim\!\sim\!\sim & test & \sim\!\sim\!\sim & achar \\
1. & \overleftrightarrow{y}_{test} & = & (1,\ 1) & +_y & (\infty,\ 0) & +_y & (1,\ 1) \\
 & & = & (\infty,\ 2) & & & & \\
2. & \overleftrightarrow{y}_{test} & = & (1,\ 1) & +_y & (\infty,\ 2) & +_y & (1,\ 1) \\
 & & = & (\infty,\ 4) & & & & \\
 & & ... & & & & & \\
 & & = & (\infty,\ \infty) & & & &
\end{array}
$$

The parser *test* does not terminate in this case.

Yield size constraint revisited

The definitions of the yield size operators $+_y$, $\cup_y$ and $\cap_y$ are very easy and convenient. However, we need to take a closer look on the operator $\cap_y$ for filter applications. Consider the following example:

$$
\begin{array}{llcccccc}
 & f & = & (achar\ empty & \sim\!\sim\!\sim & f) & \textbf{with}\ (size\ 3\ 8) & ||| \\
1. & \overleftrightarrow{y}_f & = & ((1,1)\ (0,0) & +_y & (\infty,0)) & \cap_y \quad (3,8) & \cup_y \\
 & & = & (0,1) & & & & \\
2. & \overleftrightarrow{y}_f & = & ((1,1)\ (0,0) & +_y & (0,1)) & \cap_y \quad (3,8) & \cup_y \\
 & & = & (0,2) & & & & \\
 & & ... & & & & & \\
 & & = & (0,\infty) & & & &
\end{array}
$$

Here, the yield size of nonterminal f is calculated to $(0,\infty)$, which is wrong. Since the yield parser works in a bottom-up fashion, the applications of $size\ 3\ 8$ filter out every application of $achar \sim\!\sim\!\sim f$. So the correct yield size of nonterminal f is $(0,0)$. A variant of the $\cap_y$ operator corrects this problem:

$$
\begin{array}{lcl}
\overleftrightarrow{y}_p \cap_y \overset{\cap}{y}_f & = & (l_p,\ u_p) \cap_y (l_f,\ u_f) \\
 & = & \textbf{if}\ l_r > u_r\ \textbf{then}\ (\infty,0)\ \textbf{else}\ (l_r,u_r) \\
 & & \text{where} \\
 & & \quad (l_r,u_r) = (max\ l_p\ l_f,\ min\ u_p\ u_f)
\end{array}
$$

With the new definition of the $\cap_y$ operator we get the following result for our example:

$$
\begin{array}{lrlllllll}
 & f & = & (achar & \sim\!\sim\!\sim & f) & \textbf{with} & (size\ 3\ 8) & |||\\
 & & & empty & & & & & \\
1. & \overleftrightarrow{y}_f & = & ((1,1) & +_y & (\infty,0)) & \cap_y & (3,8) & \cup_y\\
 & & & (0,0) & & & & & \\
 & & = & (0,0) & & & & & \\
2. & \overleftrightarrow{y}_f & = & ((1,1) & +_y & (0,0)) & \cap_y & (3,8) & \cup_y\\
 & & & (0,0) & & & & & \\
 & & = & (0,0) & & & & &
\end{array}
$$

This calculation results in the correct yield size of nonterminal f. However, it is an open problem whether this new definition is correct in full generality.

Abstract interpretation

The yield size analysis orients itself at the technique of abstract interpretation [4]. A classical example for abstract interpretation in the area of compiler construction is the strictness analysis of functional programs [3, 36, 38]. The major characteristic of the abstract interpretation is the abstraction of the input data domain to an abstract domain. In our application this is the domain of the yield sizes $(\mathbb{N}^\infty, \mathbb{N}^\infty)$. However, an embedding into the formalisms of the classical abstract interpretation is quite difficult. Here we do not abstract directly on the input and output of a program, but on a tree grammar. One can also think of that the yield size analysis can be an application of grammar flow analysis [32, 56]. Here however, we use a direct approach.

4.1.3 Combinator optimization

As described in Section 2.5.5, each occurrence of a $\sim\!\sim\!\sim$ combinator multiplies the runtime by the input length. An expression $p \sim\!\sim\!\sim q$ iterates over all possible subwords of p and q. When p or q have a constant or an upper bounded yield size, this is a fruitless effort.

The common approach during the development of an embedded ADP prototype is to introduce optimized variants of this combinator. For example, the parsers $r -\!\sim\!\sim p$ and $p \sim\!\sim - r$ have the same asymptotic runtime as p alone.

Accordingly, we can develop variants for other cases, or we can directly use the $\sim\!\sim$ combinator with concrete yield sizes. For larger programs, this can easily become rather complex. Moreover, defining a wrong yield size can also lead to wrong program results.

With the help of yield size analysis, we can completely automate this optimization. The process divides into two phases:

1. In the first phase all occurrences of the $\sim\!\sim\!\sim$ combinator are replaced by the yield size enriched combinator $\sim\!\sim$.

2. To increase readability, all combinators are then replaced by their specialized versions (if available).

Phase I: enrich combinators

Overview: enrich combinators
The function *enc* replaces the "standard-combinator" $\sim\!\sim\!\sim$ by the yield size-enriched combinator $\sim\!\sim$.

enc : enrich next combinators
enc $u_p \rightarrow (\overleftrightarrow{y}_p,\ u_{\bar{p}})$

Input	*Output*
u_p: Production	$\overleftrightarrow{y}_p$: Yield size of u_p
	$u_{\bar{p}}$: Optimized production p

Figure 4.1: The combinator-enrichment-function *enc*

Let p be a production with $p = u_p$. The combinator optimization is initialized with $(\overleftrightarrow{y}_p,\ u_{\bar{p}}) = enc\ u_p$. The function *enc* copies the input-structure and also carries the yield sizes. An expression $p \sim\!\sim\!\sim q$ is then replaced by the yield size enriched expression $(\bar{p},\ \overleftrightarrow{y}_p) \sim\!\sim (\bar{q},\ \overleftrightarrow{y}_q)$.

$(\overleftrightarrow{y}_p,\ u_{\bar{p}}) = enc\ u_p$ with

$$enc\ (Terminal\ t) = (\overleftrightarrow{y}_t,\ Terminal\ t)$$

The yield size $\overleftrightarrow{y}_{nt}$ of a nonterminal nt is taken directly from the results of yield size analysis:

$$\begin{aligned} enc\ (Nonterminal\ nt) &= (\overleftrightarrow{y}_{nt},\ Nonterminal\ nt) \\ enc\ (p :\!\sim\!\sim\!\sim q) &= (\overleftrightarrow{y}_p +_y \overleftrightarrow{y}_q,\ (\bar{p},\ \overleftrightarrow{y}_p) :\!\sim\!\sim (\bar{q},\ \overleftrightarrow{y}_q)) \end{aligned}$$

with

$$\begin{aligned} (\overleftrightarrow{y}_p,\ \bar{p}) &= enc\ p \\ (\overleftrightarrow{y}_q,\ \bar{q}) &= enc\ q \end{aligned}$$

With the $\sim\!\sim$ combinator, the user can declare concrete yield sizes right from the start. An expression $((p,\ \overset{\cap}{y}_{fl}) \sim\!\sim (q,\ \overset{\cap}{y}_{fr}))$ is semantically equivalent to $(p\ \textbf{with}\ fl) \sim\!\sim\!\sim (q\ \textbf{with}\ fr)$. Figure 4.2 shows an example.

A production

$$test \;=\; f \mathbin{<<<} (astringp \;\; \textbf{with} \;\; size\; 4\; 9) \;\sim\!\!\sim\!\!\sim\; (astringp \;\; \textbf{with} \;\; minsize\; 3)$$

can also be written as

$$\begin{aligned} test \;&=\; f \mathbin{<<<} astringp \sim\!\!\sim ! \; astringp \\ &\quad\; \textbf{where} \\ &\qquad (\sim\!\!\sim !) \;=\; (\sim\!\!\sim *)\; (4,\; 9)\; 3 \end{aligned}$$

The combinator $\sim\!\!\sim *$ is a special case of the $\sim\!\!\sim$ combinator for an unlimited parser on the right side.

Figure 4.2: Semantics of the $\sim\!\!\sim$ combinator

However, the case can arise here, that the user defined $\overset{\cap}{y}_p$ and $\overset{\cap}{y}_q$ collide with the values calculated in yield size analysis. For example, the production

$$\begin{aligned} test \;&=\; f \mathbin{<<<} (astringp \;\; \textbf{with} \;\; maxsize\; 3) \sim\!\!\sim ! \; astringp \\ &\quad\; \textbf{where} \\ &\qquad (\sim\!\!\sim !) \;=\; (\sim\!\!\sim *)\; (5,\; 10)\; 1 \end{aligned}$$

makes no sense, since $\overset{\leftrightarrow}{y}_p \;\cap_y\; \overset{\cap}{y}_p = (1,\; 3) \;\cap_y\; (5,\; 10) = (5,\; 3)$. In this case, the analysis ends with an error message:

$$enc\; ((p,\; \overset{\cap}{y}_p) :\!\sim\!\sim (q,\; \overset{\cap}{y}_q))$$

$$= \begin{cases} error\; \text{“}yield\; size\; conflict\text{”} & \text{if } \; l_{\bar{p}} > u_{\bar{p}} \vee l_{\bar{q}} > u_{\bar{q}} \\ (\overset{\leftrightarrow}{y}_{\bar{p}} \;+_y\; \overset{\leftrightarrow}{y}_{\bar{q}},\; (\bar{p},\; \overset{\leftrightarrow}{y}_{\bar{p}}) :\!\sim\!\sim (\bar{q},\; \overset{\leftrightarrow}{y}_{\bar{q}})) & \text{otherwise} \end{cases}$$

with

$$\begin{aligned} (\overset{\leftrightarrow}{y}_p,\; \bar{p}) \;&=\; enc\; p \\ (\overset{\leftrightarrow}{y}_q,\; \bar{q}) \;&=\; enc\; q \\ \overset{\leftrightarrow}{y}_{\bar{p}} \;&=\; \overset{\leftrightarrow}{y}_p \;\cap_y\; \overset{\cap}{y}_p \\ \overset{\leftrightarrow}{y}_{\bar{q}} \;&=\; \overset{\leftrightarrow}{y}_q \;\cap_y\; \overset{\cap}{y}_q \end{aligned}$$

$$\begin{aligned} enc\; (c :\!<<< p) \;&=\; (\overset{\leftrightarrow}{y}_p,\; c :\!<<< \bar{p}) \\ \text{with } (\overset{\leftrightarrow}{y}_p,\; \bar{p}) \;&=\; enc\; p \end{aligned}$$

$$
\begin{array}{lll}
enc\ (p\ :|||\ q) & = & (\overleftrightarrow{y}_p\ \cup_y\ \overleftrightarrow{y}_q,\ \bar{p}\ :|||\ \bar{q}) \\
\quad \text{with} & & \\
\qquad (\overleftrightarrow{y}_p,\ \bar{p}) & = & enc\ p \\
\qquad (\overleftrightarrow{y}_q,\ \bar{q}) & = & enc\ q
\end{array}
$$

During the transformation, we can omit all length-limited filter applications. For example, the expression

$$astringp\ \sim\!\sim\!\sim\ (astringp\ \textbf{with}\ minsize\ 3)$$

is transformed into the expression

$$(astringp,\ (1,\ \infty))\ \sim\!\sim\ ((astringp\ \textbf{with}\ minsize\ 3),\ (3,\ \infty))$$

which makes the additional filter application *minsize* 3 obsolete. However, to increase readability of the transformed program, we will keep this additional filter. Here the runtime is only increased by a small constant factor.

$$
\begin{array}{lll}
enc\ (p\ \text{`With`}\ f) & = & (\overleftrightarrow{y}_p\ \cap_y\ \overset{\cap}{y}_f,\ \bar{p}\ \text{`With`}\ f) \\
\quad \text{with}\ (\overleftrightarrow{y}_p,\ \bar{p}) & = & enc\ p \\
 & & \\
enc\ (p\ :\ldots\ h) & = & (\overleftrightarrow{y}_p,\ \bar{p}\ :\ldots\ h) \\
\quad \text{with}\ (\overleftrightarrow{y}_p,\ \bar{p}) & = & enc\ p
\end{array}
$$

Figure 4.3 shows an example.

```
p1 = tabulated (
     ((f <<< ((string ``Hello'') ~~~ (string ``world''))) |||
     (r <<< ((achar ~~~ astring) ~~~ astringp))))
```

```
-- production p1, yield size: (2,Infinite)
p1 = tabulated (
     ((f <<< ((string ``Hello'') ((~~) (5,5) (5,5)) (string ``world''))) |||
     (r <<< ((achar ((~~) (1,1)        (0,Infinite)) astring)
                    ((~~) (1,Infinite) (1,Infinite)) astringp))))
```

Figure 4.3: Example for combinator optimization

Phase II: replace combinators

After this optimization phase, we can now directly print out the optimized version of the program. Due to the exclusive use of the $\sim\!\sim$ combinator, this version is not very readable.

In a second transformation phase, we will now transform the $\sim\!\!\sim$ combinators into a set of predefined special combinators. The following table gives an overview:

$\overleftrightarrow{y}_p$	$\overleftrightarrow{y}_q$	Variant
$(0,\ \infty)$	$(0,\ \infty)$	$\sim\!\!\sim\!\!\sim$
$(1,\ \infty)$	$(0,\ \infty)$	$+\!\sim\!\!\sim$
$(0,\ \infty)$	$(1,\ \infty)$	$\sim\!\!\sim\!+$
$(1,\ \infty)$	$(1,\ \infty)$	$\sim\!-\!\sim$
$(1,\ 1)$	$(0,\ \infty)$	$-\!\sim\!\!\sim$
$(0,\ \infty)$	$(1,\ 1)$	$\sim\!\!\sim\!-$

For all other cases we generate a combinator of the form $\sim\!\!\sim!^{+} = (\sim\!\!\sim)\ \overleftrightarrow{y}_p\ \ \overleftrightarrow{y}_q$, where the number of "!" depends on the number of generated combinators in the current production. As shown in Figure 4.2 we use the following special cases of $\sim\!\!\sim$ for unlimited yield sizes:

$$
\begin{array}{llcllll}
(\sim\!\!\sim\!*) & \overleftrightarrow{y}_p & l_q & = & (\sim\!\!\sim) & \overleftrightarrow{y}_p & (l_q,\ \infty) \\
(*\!\sim\!\!\sim) & l_p & \overleftrightarrow{y}_q & = & (\sim\!\!\sim) & (l_p,\ \infty) & \overleftrightarrow{y}_q \\
(*\!\sim\!*) & l_p & l_q & = & (\sim\!\!\sim) & (l_p,\ \infty) & (l_q,\ \infty)
\end{array}
$$

Figure 4.4 shows the result for the example of Figure 4.3.

```
-- production p1, yield size: (2,Infinite)
p1 = tabulated (
     ((f <<< ((string ‘‘Hello’’) ((~~) (5,5) (5,5)) (string ‘‘world’’))) |||
     (r <<< ((achar ((~~) (1,1)        (0,Infinite)) astring)
                    ((~~) (1,Infinite) (1,Infinite)) astringp))))
```

```
-- production p1, yield size: (2,Infinite)
p1 = tabulated (
     ((f <<< ((string ‘‘Hello’’) ~~! (string ‘‘world’’)))  |||
      (r <<< ((achar -~~ astring) ~-~ astringp))))
      where
        (~~!) = (~~) (5,5) (5,5)
```

Figure 4.4: Example for a replacement of combinators

4.2 Recurrence derivation

No subscripts, no errors!

One of the major strengths of Algebraic Dynamic Programming is the lack of subscripts. Since no subscripts are used, no subscript errors can be made. In traditional dynamic programming, subscript errors are a frequent source of error.

In the implementation of ADP programs in an imperative language, we can hardly do without subscripts. In the following section we describe an approach to calculate the subscripts for a given ADP program.

4.2.1 Subword convention

A central property of Algebraic Dynamic Programming is the access to subwords of the input sequence. Let x be a word of length n with $x = x_1...x_n$. The subscript pair $(i,\ j)$ stands for a subword of x with $(i,\ j) = x_{i+1}...x_j$. This results in a number of interesting properties:

$$\begin{aligned} (0,\ n) &= x \\ (0,\ k) +\!\!+ (k,\ n) &= x \\ |(i,\ j)| &= j - i \quad \text{with } i \leq j \end{aligned}$$

Example: ${}_0H_1E_2L_3L_4O_5$

This property will be helpful during the subscript calculation.

4.2.2 Intermediate language

Until now, we have described all transformation on the abstract syntax of the source language (Section 3.1). For the subscript derivation we now move to an intermediate language

$\mathcal{S}$. We describe this intermediate language by grammar $\mathcal{G}_{\mathcal{S}}$:

$$
\begin{aligned}
\textbf{data}\ \text{Production}_{\mathcal{S}} &= \text{Production}_{\mathcal{S}}\ \mathit{Ident}\ \text{Unit}_{\mathcal{S}} \\
\textbf{data}\ \text{Unit}_{\mathcal{S}} &= \mathit{Terminal}_{\mathcal{S}}\ \mathit{Ident}\ \text{Subscripts} \\
&\mid \mathit{Nonterminal}_{\mathcal{S}}\ \mathit{Ident}\ \text{Subscripts} \\
&\mid \text{Unit}_{\mathcal{S}}\ :\sim\sim\sim_{\mathcal{S}}\ \text{Unit}_{\mathcal{S}} \\
&\mid \text{Function}\ :<<<_{\mathcal{S}}\ \text{Unit}_{\mathcal{S}} \\
&\mid \text{Unit}_{\mathcal{S}}\ \text{`With`}_{\mathcal{S}}\ (\mathit{Ident}, \text{Subscripts}) \\
&\mid \text{Unit}_{\mathcal{S}}\ :|||_{\mathcal{S}}\ \text{Unit}_{\mathcal{S}} \\
&\mid \text{Unit}_{\mathcal{S}}\ :\ldots_{\mathcal{S}}\ \mathit{Ident}
\end{aligned}
$$

Unlike in the abstract syntax of the source language, the two variants of the $\sim\sim\sim$ combinator ($\sim\sim\sim$ and $\sim\sim$) exist only in one variant here. We will eliminate the $\sim\sim$ combinator in the method described below. Additionally, terminal symbols, nonterminal symbols and filters are enriched with an expression for the subscripts:

$$
\begin{aligned}
\textbf{type}\ \text{Subscripts} &= (\text{MathExp}, \text{MathExp}) \\
\textbf{data}\ \text{MathExp} &= \mathit{MNum}\ \mathit{Int} \\
&\mid \mathit{MVar}\ \mathit{Ident} \\
&\mid \text{MathExp}\ :+\ \text{MathExp} \\
&\mid \text{MathExp}\ :-\ \text{MathExp} \\
&\mid \mathit{Max}\ \text{MathExp}\ \text{MathExp} \\
&\mid \mathit{Min}\ \text{MathExp}\ \text{MathExp}
\end{aligned}
$$

The application of an algebra function is also extended by additional information. The field Bounds is used in later translation phases for checking the input length of the input. Loop contains information about the generated inner loops in the imperative representation:

$$
\begin{aligned}
\textbf{type}\ \text{Function} &= (\mathit{Ident}, \text{Bounds}, [\text{Loop}]) \\
\textbf{type}\ \text{Bounds} &= (\text{Subscripts}, \mathit{YSize}) \\
\textbf{type}\ \text{Loop} &= (\mathit{Ident}, \text{MathExp}, \text{MathExp})
\end{aligned}
$$

4.2.3 Phase I

For a simplified start in the description of the procedure, we will restrict the analysis to terminal parsers combined with the $\sim\sim\sim$ combinator.

Let p be a production with $p = u_p$. We assume that we will generate a double-loop of the form

Overview: Subscript derivation (1)
The function *dss* calculates the subscripts and transforms the source language into the language $\mathcal{S}$.

dss : derive subscripts
$dss\ (i,\ j)\ u_p \rightarrow (\overleftrightarrow{y}_p,\ u_p^{\mathcal{S}})$

Input	*Output*
$(i,\ j)$: Subscript expressions	$\overleftrightarrow{y}_p$: Yield size of u_p
u_p: Production	$u_p^{\mathcal{S}}$: Subscript-enriched production u_p

Figure 4.5: The subscript derivation *dss* (simplified)

for $j := 0$ **to** n **do**
  for $i := j$ **to** 0 **do**
    $u_p^{\mathcal{S}}\ (i,\ j)$

in the imperative target program. Let $u_p^{\mathcal{S}}$ be the subscript-enriched version of the production u_p. We initialize the procedure with $dss\ (i,\ j)\ u_p$. Corresponding to Section 4.1.3, we will carry the yield sizes together with the target structure.

$$
\begin{array}{lll}
dss\ (i,\ j)\ (Terminal\ t) & = & (\overleftrightarrow{y}_t,\ Terminal_{\mathcal{S}}\ t\ (i,\ j)) \\
dss\ (i,\ j)\ (p :\sim\sim\sim\ q) & = & (\overleftrightarrow{y}_p\ +_y\ \overleftrightarrow{y}_q,\ \bar{p} :\sim\sim\sim_{\mathcal{S}}\ \bar{q}) \\
\quad \text{with} & & \\
\quad (\overleftrightarrow{y}_p,\ \bar{p}) & = & dss\ (i,\ j_p)\ \ p \\
\quad (\overleftrightarrow{y}_q,\ \bar{q}) & = & dss\ (i_q,\ j)\ \ q
\end{array}
$$

For the subscript variables j_p and i_q we now need to examine three alternative cases. If p has a constant yield size, we can clearly set j_p to $i + l_p$. Correspondingly, let $i_q = j - l_q$ for a constant yield size of q. If neither p nor q have a constant yield size, an inner loop has to be introduced. In this case we use a loop variable k. Due to the subword convention, we

can set $j_p = i_q$ in all other cases. Let $cnst\ \overleftrightarrow{y}_p\ =\ cnst\ (l_p,\ u_p)\ =\ l_p \equiv u_p$.

$$j_p = \begin{cases} k & \text{for } \neg\ cnst\ \overleftrightarrow{y}_p \wedge \neg\ cnst\ \overleftrightarrow{y}_q \\ i + l_p & \text{for } cnst\ \overleftrightarrow{y}_p \\ i_q & \text{otherwise} \end{cases}$$

$$i_q = \begin{cases} k & \text{for } \neg\ cnst\ \overleftrightarrow{y}_p \wedge \neg\ cnst\ \overleftrightarrow{y}_q \\ j - l_q & \text{for } cnst\ \overleftrightarrow{y}_q \\ j_p & \text{otherwise} \end{cases} \tag{4.4}$$

The loop boundaries of an inner loop variable k are calculated from the overlap of the two parsers p and q. This is depicted in Figure 4.6. One shall also note the direct correspondence to the definitions of the different variants of the $\sim\!\!\sim$ combinator (Figure 4.7):

$$k_{start} = \begin{cases} i + l_p & \text{for } u_q \equiv \infty \\ max\ (i + l_p)\ (j - u_q) & \text{otherwise} \end{cases}$$

$$k_{end} = \begin{cases} j - l_q & \text{for } u_p \equiv \infty \\ min\ (i + u_p)\ (j - l_q) & \text{otherwise} \end{cases} \tag{4.5}$$

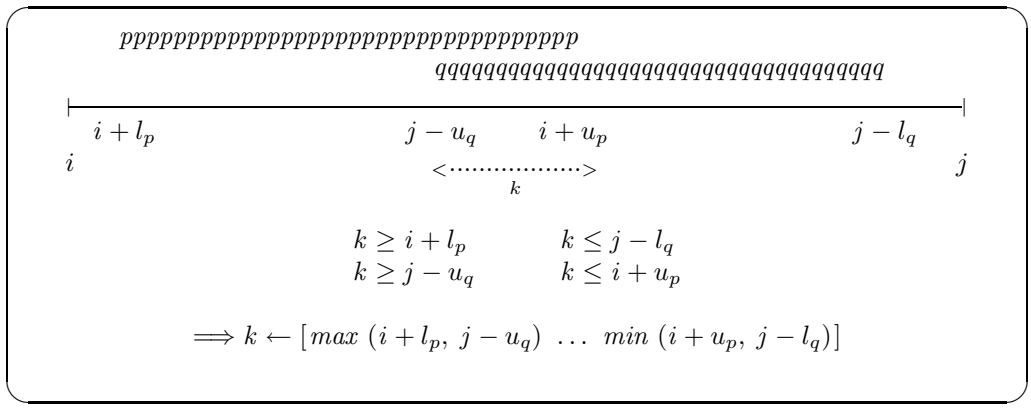


Figure 4.6: Running boundaries of a loop k

$$
\begin{array}{l}
(\sim\!\sim) :: (Int,\ Int) \to (Int,\ Int) \to Parser\ (a \to b) \to Parser\ a \to Parser\ b \\
(\sim\!\sim)\ (l_p,\ u_p)\ (l_q,\ u_q)\ p\ q\ (i,\ j) \\
\qquad = [x\ y \mid k \leftarrow [max\ (i + l_p)\ (j - u_q)\,..\,min\ (i + u_p)\ (j - l_q)], \\
\qquad\quad x \leftarrow p\ (i,\ k),\ y \leftarrow q\ (k,\ j)] \\
\\
(\sim\!\sim\!*) :: (Int,\ Int) \to Int \to Parser\ (a \to b) \to Parser\ a \to Parser\ b \\
(\sim\!\sim\!*)\ (l_p,\ u_p)\ l_q\ p\ q\ (i,\ j) \\
\qquad = [x\ y \mid k \leftarrow [(i + l_p)\,..\,min\ (i + u_p)\ (j - l_q)], \\
\qquad\quad x \leftarrow p\ (i,\ k),\ y \leftarrow q\ (k,\ j)] \\
\\
(*\!\sim\!\sim) :: Int \to (Int,\ Int) \to Parser\ (a \to b) \to Parser\ a \to Parser\ b \\
(*\!\sim\!\sim)\ l_p\ (l_q,\ u_q)\ p\ q\ (i,\ j) \\
\qquad = [x\ y \mid k \leftarrow [max\ (i + l_p)\ (j - u_q)\,..\,(j - l_q)], \\
\qquad\quad x \leftarrow p\ (i,\ k),\ y \leftarrow q\ (k,\ j)] \\
\\
(*\!\sim\!*) :: Int \to Int \to Parser\ (a \to b) \to Parser\ a \to Parser\ b \\
(*\!\sim\!*)\ l_p\ l_q\ p\ q\ (i,\ j) \\
\qquad = [x\ y \mid k \leftarrow [(i + l_p)\,..\,(j - l_q)], \\
\qquad\quad x \leftarrow p\ (i,\ k),\ y \leftarrow q\ (k,\ j)]
\end{array}
$$

Figure 4.7: Combinators with concrete length declarations

Example 1. Let p be a production with $p = achar\ \sim\!\sim\!\sim\ astringp$. The subscripts of p are calculated as follows:

$$
\begin{array}{lll}
dss\ (i,\ j)\ (achar\ :\!\sim\!\sim\!\sim\ astringp) & = (\overleftrightarrow{y}_p\ +_y\ \overleftrightarrow{y}_q,\ \bar{p}\ :\!\sim\!\sim\!\sim_{\mathcal{S}}\ \bar{q}) \\
\quad \text{with} & \\
\\
(\overleftrightarrow{y}_p,\ \bar{p}) = dss\ (i,\ j_p)\ achar & = ((1,\ 1),\ achar\ (i,\ j_p)) \\
(\overleftrightarrow{y}_q,\ \bar{q}) = dss\ (i_q,\ j)\ astringp & = ((1,\ \infty),\ astringp\ (i_q,\ j)) \\
\\
= ((2,\ \infty),\ achar\ (i,\ j_p)\ :\!\sim\!\sim\!\sim_{\mathcal{S}}\ astringp\ (i_q,\ j)) &
\end{array}
$$

Following equation 4.4, $j_p = i + l_p = i + 1$ and $i_q = j_p = i + 1$ hold. Therefore:

$$
\begin{array}{l}
dss\ (i,\ j)\ (achar\ :\!\sim\!\sim\!\sim\ astringp) \\
\quad = ((2,\ \infty),\ achar\ (i,\ i + 1)\ :\!\sim\!\sim\!\sim_{\mathcal{S}}\ astringp\ (i + 1,\ j))
\end{array}
$$

Example 2. For this example, let $p = astringp \sim\sim\sim astringp$:

$$dss\ (i,\ j)\ (astringp\ :\sim\sim\sim\ astringp) \quad = (\overleftrightarrow{y}_p\ +_y\ \overleftrightarrow{y}_q,\ \bar{p}\ :\sim\sim\sim_{\mathcal{S}}\ \bar{q})$$

with

$$(\overleftrightarrow{y}_p,\ \bar{p}) = dss\ (i,\ j_p)\ astringp \quad = ((1,\ \infty),\ astringp\ (i,\ j_p))$$
$$(\overleftrightarrow{y}_q,\ \bar{q}) = dss\ (i_q,\ j)\ astringp \quad = ((1,\ \infty),\ astringp\ (i_q,\ j))$$

$$= ((2,\ \infty),\ astringp\ (i,\ j_p)\ :\sim\sim\sim_{\mathcal{S}}\ astringp\ (i_q,\ j))$$

Since neither $\overleftrightarrow{y}_p$ nor $\overleftrightarrow{y}_q$ are constant, a new loop has to be introduced. The loop boundaries are calculated in correspondence to Equation 4.5 to $k_{start} = i + l_p = i + 1$ and $k_{end} = j - l_q = j - 1$. Therefore:

$$dss\ (i,\ j)\ (astringp\ :\sim\sim\sim\ astringp)$$
$$= ((2,\ \infty),\ astringp\ (i,\ k)\ :\sim\sim\sim_{\mathcal{S}}\ astringp\ (k,\ j)) \qquad \text{with } k \leftarrow [\,i+1\ \ldots\ j-1]$$

4.2.4 Yield size constraint

Overview: Subscript derivation (2)
The function *dss* calculates the subscripts and transforms the source language into the language $\mathcal{S}$.

dss : derive subscripts
$dss\ (i,\ j)\ \overset{\cap}{y}_U\ u_p \rightarrow (\overleftrightarrow{y}_p,\ u_p^{\mathcal{S}})$

Input	*Output*
$(i,\ j)$: Subscript expressions	$\overleftrightarrow{y}_p$: Yield size of u_p
u_p: Production	$u_p^{\mathcal{S}}$: Subscript-enriched production u_p
$\overset{\cap}{y}_U$: Yield size constraint for u_p	

Figure 4.8: The subscript calculation function *dss*

For the calculation of filter expressions it is necessary to take the yield size constraints into account. The application of a filter f on an expression q with yield size $\overleftrightarrow{y}_q$ results in $\overleftrightarrow{y}_q\ \cap_y\ \overset{\cap}{y}_f$ as the new yield size for q. Therefore, we extend the function *dss* with an additional argument $\overset{\cap}{y}_U$ for the yield size constraint. This constraint will be initialized with $(0,\ \infty)$.

$$
\begin{aligned}
&dss\ (i,\ j)\ \overset{\cap}{y}_U\ \ (Terminal\ t) &&= (\overset{\leftrightarrow}{y}_t\ \cap_y\ \overset{\cap}{y}_U,\ Terminal_{\mathcal{S}}\ t\ (i,\ j))\\
&dss\ (i,\ j)\ \overset{\cap}{y}_U\ \ (Nonterminal\ nt) &&= (\overset{\leftrightarrow}{y}_{nt}\ \cap_y\ \overset{\cap}{y}_U,\ Nonterminal_{\mathcal{S}}\ nt\ (i,\ j))
\end{aligned}
$$

Filter expression can also be subject to a yield size constraint. Therefore, $(p\ \ \text{with}\ \ f)$ is updated with $\overset{\cap}{y}_f\ \cap_y\ \overset{\cap}{y}_U$:

$$
\begin{aligned}
&dss\ (i,\ j)\ \overset{\cap}{y}_U\ \ (p\ \text{`With`}\ f)\ =\ (\overset{\leftrightarrow}{y}_p,\ \bar{p}\ \text{`With`}_{\mathcal{S}}\ \ (f,\ (i,\ j)))\\
&\quad \text{with}\\
&\quad (\overset{\leftrightarrow}{y}_p,\ \bar{p})\ =\ dss\ (i,\ j)\ (\overset{\cap}{y}_f\ \cap_y\ \overset{\cap}{y}_U)\ p
\end{aligned}
$$

For the $\sim\!\sim$ combinator, the user-defined $\overset{\cap}{y}_p$ and $\overset{\cap}{y}_q$ are passed to p and q. The subscript variables j_p and i_q are calculated as in Equation 4.4.

$$
\begin{aligned}
&dss\ (i,\ j)\ \overset{\cap}{y}_U\ \ ((p,\ \overset{\cap}{y}_p) :\!\sim\!\sim (q,\ \overset{\cap}{y}_q))\ =\ (\overset{\leftrightarrow}{y}_U,\ \bar{p} :\!\sim\!\sim\!\sim_{\mathcal{S}} \bar{q})\\
&\quad \text{with}\\
&\quad (\overset{\leftrightarrow}{y}_p,\ \bar{p}) &&=\ dss\ (i,\ j_p)\ \overset{\cap}{y}_p\ \ p\\
&\quad (\overset{\leftrightarrow}{y}_q,\ \bar{q}) &&=\ dss\ (i_q,\ j)\ \overset{\cap}{y}_q\ \ q\\
&\quad \overset{\leftrightarrow}{y}_U &&=\ (\overset{\leftrightarrow}{y}_p\ +_y\ \overset{\leftrightarrow}{y}_q)\ \cap_y\ \overset{\cap}{y}_U
\end{aligned}
$$

The $\sim\!\sim\!\sim$ combinator now is a special case of the $\sim\!\sim$ combinator:

$$
dss\ (i,\ j)\ \overset{\cap}{y}_U\ \ (p :\!\sim\!\sim\!\sim q)\ =\ dss\ (i,\ j)\ \overset{\cap}{y}_U\ \ ((p,\ (0,\ \infty)) :\!\sim\!\sim (q,\ (0,\ \infty)))
$$

The loops $loops_p$ collected during processing of expression p are tied to the algebra function application f. Additionally the current subscripts $(i,\ j)$ are saved together with the yield size $\overset{\leftrightarrow}{y}_p$ to allow length checks in the target language.

$$
\begin{aligned}
&dss\ (i,\ j)\ \overset{\cap}{y}_U\ \ (f :\!<\!<\!< p)\ =\ (\overset{\leftrightarrow}{y}_p,\ (f,\ ((i,\ j),\ \overset{\leftrightarrow}{y}_p),\ loops_p) :\!<\!<\!<_{\mathcal{S}} \bar{p})\\
&\quad \text{with}\\
&\quad (\overset{\leftrightarrow}{y}_p,\ \bar{p}) &&=\ dss\ (i,\ j)\ \overset{\cap}{y}_U\ \ p
\end{aligned}
$$

Expressions combined via the $|||$ combinator are processed symmetrically:

$$
\begin{aligned}
&dss\ (i,\ j)\ \overset{\cap}{y}_U\ \ (p\ :|||\ \ q)\ =\ (\overset{\leftrightarrow}{y}_U,\ (\bar{p}\ :|||_{\mathcal{S}}\ \ \bar{q}))\\
&\quad \text{with}\\
&\quad (\overset{\leftrightarrow}{y}_p,\ \bar{p}) &&=\ dss\ (i,\ j)\ \overset{\cap}{y}_U\ \ p\\
&\quad (\overset{\leftrightarrow}{y}_q,\ \bar{q}) &&=\ dss\ (i,\ j)\ \overset{\cap}{y}_U\ \ q\\
&\quad \overset{\leftrightarrow}{y}_U &&=\ \overset{\leftrightarrow}{y}_p\ \cup_y\ \overset{\leftrightarrow}{y}_q
\end{aligned}
$$

Finally, the calculation for the choice combinator:

$$
\begin{array}{l}
dss\ (i,\ j)\ \overset{\cap}{y}_U\ \ (p\ \ :\ldots\ h)\ =\ (\overleftrightarrow{y}_p,\ (\bar{p}\ \ :\ldots_{\mathcal{S}}\ h)) \\
\quad \text{with} \\
\quad (\overleftrightarrow{y}_p,\ \bar{p}) \quad\ =\ dss\ (i,\ j)\ \overset{\cap}{y}_U\ \ p
\end{array}
$$

Example 3. We will now demonstrate how the yield size constraints behave during the subscript calculation. For this, we extend the production of Example 2 with the length-limiting filter *maxsize* 7. Let $p\ =\ astringp\ \sim\sim\sim\ (astringp\ \text{with}\ (maxsize\ 7))$.

$$
\begin{array}{l}
dss\ (i,\ j)\ (0,\ \infty)\ (astringp\ :\sim\sim\sim\ (astringp\ \text{`With`}\ (maxsize\ 7)))\ = \\
dss\ (i,\ j)\ (0,\ \infty)\ ((astringp,\ (0,\ \infty)):\sim\sim((astringp\ \text{`With`}\ (maxsize\ 7)),\ (0,\ \infty)))\ = \\
\quad (\overleftrightarrow{y}_p\ +_y\ \overleftrightarrow{y}_q,\ \bar{p}\ :\sim\sim\sim_{\mathcal{S}}\ \bar{q}) \\
\quad \text{with}
\end{array}
$$

$$
\begin{array}{lcl}
(\overleftrightarrow{y}_p,\ \bar{p}) & = & dss\ (i,\ j_p)\ (0,\ \infty)\ astringp\ =\ ((1,\ \infty),\ astringp\ (i,\ j_p)) \\
(\overleftrightarrow{y}_q,\ \bar{q}) & = & dss\ (i_q,\ j)\ (0,\ \infty)\ (astringp\ \text{`With`}\ (maxsize\ 7)) \\
 & = & ((1,\ \infty)\ \cap_y\ (0,\ 7),\ astringp\ (i_q,\ j)\ \text{`With`}_{\mathcal{S}}\ ((maxsize\ 7)\ (i_q,\ j))) \\
 & = & ((1,\ 7),\ astringp\ (i_q,\ j)\ \text{`With`}_{\mathcal{S}}\ ((maxsize\ 7)\ (i_q,\ j)))
\end{array}
$$

Again, neither $\overleftrightarrow{y}_p$ nor $\overleftrightarrow{y}_q$ are constant. Corresponding to Equation 4.5, the loop boundaries are calculated to $k_{start}\ =\ max\ (i+l_p)\ (j-u_q)\ =\ max\ (i+1)\ (j-7)$ and $k_{end}\ =\ j-l_q\ =\ j-1$. Therefore:

$$
\begin{array}{l}
dss\ (i,\ j)\ (0,\ \infty)\ (astringp\ :\sim\sim\sim\ (astringp\ \text{`With`}\ (maxsize\ 7))) \\
\quad =\ ((2,\ \infty),\ astringp\ (i,\ k)\ :\sim\sim\sim_{\mathcal{S}}\ (astringp\ (k,\ j)\ \text{`With`}_{\mathcal{S}}\ ((maxsize\ 7)\ (k,\ j)))) \\
\qquad \text{with}\ k \leftarrow [max\ (i+1)\ (j-7)\ \ldots\ j-1]
\end{array}
$$

4.3 Dependency analysis

Before we generate the target program, the calculation order of the recurrences needs to be derived. It must be guaranteed that all values are calculated before they are used. In functional programming languages with lazy evaluation this is an integral part of the evaluation model. In imperative programming languages however, we need to take care of the order of calculation.

In a recurrence, all data access is restricted to results of subwords of the same or of smaller lengths. With the construction of the outer loops, we can guarantee that results are calculated with increasing subword length. Therefore, for the dependency analysis it suffices to search for expressions that access results of the same subword.

Overview: Dependency analysis
The function *da* derives the dependencies between two productions.

da : dependency analysis
$da\ q\ (l_l,\ l_r)\ \overset{\cap}{y}_p\ u_p \rightarrow (\overleftrightarrow{y}_p,\ p \rightarrow q)$

Input	*Output*
q: Searched nonterminal	$\overleftrightarrow{y}_p$: Yield size of p
l_l: Left yield size	$p \rightarrow q$: Dependency between p and q
l_r: Right yield size	
$\overset{\cap}{y}_p$: Yield size constraint for the production p	
u_p: Production	

Figure 4.9: The dependency analysis function *da*

Figure 4.10 shows an example. From the results of subscript derivation it is immediately clear, that $p3$ in any case needs to be calculated *before* $p1$ and $p2$, since both use $p3!(i,j)$ in the same iteration. The usage of $p1$ inside of $p2$ requires a deeper analysis of the corresponding loops, but here also exists a dependency. All other cases are irrelevant for the dependency analysis. With a sort of these dependencies we come to the calculation order $p3, p1, p2$.

Let $p \rightarrow q$ denote the usage of a production q from a production p inside the same loop iteration. This implies that q needs to be calculated before p. The analysis works in the same manner as the subscript derivation. Instead of subscripts we now calculate and pass through the left and the right yield sizes l_l and l_r. These are used in case of a nonterminal $\bar{nt}$, to check whether an access can take place in the same loop iteration, i.e. in the form $\bar{nt}\,!\,(i,\ j)$.

Let p be given by $p = u_p$.

$p \rightarrow q$ iff $da\ q\ (0,\ 0)\ (0,\ \infty)\ u_p$ with

$$da\ nt\ (l_l,\ l_r)\ \overset{\cap}{y}_U\ (Terminal\ t) = (\overleftrightarrow{y}_t\ \cap_y\ \overset{\cap}{y}_U,\ False)$$

$$da\ nt\ (l_l,\ l_r)\ \overset{\cap}{y}_U\ (Nonterminal\ \bar{nt}) = (\overleftrightarrow{y}_{\bar{nt}}\ \cap_y\ \overset{\cap}{y}_U,\ nt \equiv \bar{nt} \wedge l_l \equiv 0 \wedge l_r \equiv 0)$$

$$da\ nt\ (l_l,\ l_r)\ \overset{\cap}{y}_U\ ((p,\ \overset{\cap}{y}_p) :\sim\sim (q,\ \overset{\cap}{y}_q)) = (\overleftrightarrow{y}_U,\ d_p \vee d_q)$$

with

```
p1 = tabulated(f1 <<< achar ~~~ p p2 ~~~ achar        |||
               f2 <<< empty                           |||
               p p3)

p2 = tabulated(f3 <<< astring ~~~ p p1 ~~~ astring    |||
               p p3)

p3 = tabulated(f4 <<< achar   ~~~ p p1)
```

Subscript derivation:

```
p1!(i, j) =

   f1(achar(i, i+1), p2!(i+1, j-1), achar(j-1, j))
   ++
   f2(empty(i, j))
   ++
   p3!(i, j)

p2!(i, j) =

   for k2 = i to j do
      for k = i to k2 do
         f3(astring(i, k), p1!(k, k2), astring(k2, j))
   ++
   p3!(i, j)

p3!(i, j) =

   f4(achar(i, i+1), p1!(i+1, j))
```

Dependencies:

→	p1	p2	p3
p1		x	
p2			
p3	x	x	

⇒ Calculation order: p3, p1, p2

Figure 4.10: Example for a dependency analysis.

$$
\begin{array}{lcl}
(\overleftrightarrow{y}_p,\ d_p) & = & da\ nt\ (l_l,\ l_r + l_q)\ \overset{\frown}{y}_p\ p \\
(\overleftrightarrow{y}_q,\ d_q) & = & da\ nt\ (l_l + l_p,\ l_r)\ \overset{\frown}{y}_q\ q \\
\overleftrightarrow{y}_U & = & (\overleftrightarrow{y}_p\ +_y\ \overleftrightarrow{y}_q)\ \cap_y\ \overset{\frown}{y}_U
\end{array}
$$

$$
\begin{array}{lcl}
da\ nt\ (l_l,\ l_r)\ \overset{\frown}{y}_U\ (p\ :\sim\sim\sim\ q) & = & da\ nt\ (l_l,\ l_r)\ \overset{\frown}{y}_U\ ((p,\ (0,\ \infty))\ :\sim\sim\ (q,\ (0,\ \infty))) \\
da\ nt\ (l_l,\ l_r)\ \overset{\frown}{y}_U\ (f\ :<<<\ p) & = & da\ nt\ (l_l,\ l_r)\ \overset{\frown}{y}_U\ p \\
da\ nt\ (l_l,\ l_r)\ \overset{\frown}{y}_U\ (p\ \text{`With`}\ f) & = & da\ nt\ (l_l,\ l_r)\ (\overset{\frown}{y}_f\ \cap_y\ \overset{\frown}{y}_U)\ p
\end{array}
$$

$$
\begin{array}{lcl}
da\ nt\ (l_l,\ l_r)\ \overset{\frown}{y}_U\ (p\ :|||\ q) & = & (\overleftrightarrow{y}_p\ \cup_y\ \overleftrightarrow{y}_q,\ d_p \vee d_q) \\
\quad \text{with} & & \\
\quad (\overleftrightarrow{y}_p,\ d_p) & = & da\ nt\ (l_l,\ l_r)\ \overset{\frown}{y}_U\ p \\
\quad (\overleftrightarrow{y}_q,\ d_q) & = & da\ nt\ (l_l,\ l_r)\ \overset{\frown}{y}_U\ q \\
da\ nt\ (l_l,\ l_r)\ \overset{\frown}{y}_U\ (p\ :\ldots\ f) & = & da\ nt\ (l_l,\ l_r)\ \overset{\frown}{y}_U\ p
\end{array}
$$

The calculation order is then determined by a topological sort on $\rightarrow$. Let P be the set of productions and D a list of already sorted productions.

$$
\begin{array}{l}
D\ =\ [\,] \\
\text{while } P \neq \{\,\} \\
\quad R\ =\ \{p \mid p,\ q \in P,\ p \not\rightarrow q\} \\
\quad D\ =\ D +\!\!+ R \\
\quad P\ =\ P \backslash R
\end{array}
$$

In each iteration of the loop the list R holds all productions that do not depend on any other production. R is then added to the list of sorted productions (D) and removed from the list P. This is repeated until the list P is empty. In case of productions with circular dependencies this procedure will not terminate. By an additional check for $R = \{\}$ we can catch this property and generate a suitable error message.

4.4 Code generation

In the following we describe the last steps of the translation: the code generation. This is done in a multi-phase process. In the first step we translate the intermediate language $\mathcal{S}$ into a second intermediate language $\mathcal{LC}$, the language of list comprehensions. The programs in this language are then optimized by eliminating unnecessary list constructs. After this optimization we translate the language $\mathcal{LC}$ into the target language $\mathcal{TL}$. These translations are summarized as follows:

$\mathcal{S} \rightarrow \mathcal{LC}$	compile to list comprehensions
$\mathcal{LC} \rightarrow \mathcal{LC}$	list elimination
$\mathcal{LC} \rightarrow \mathcal{TL}$	target code generation

4.4.1 Intermediate language for list comprehensions

As intermediate language $\mathcal{LC}$ we use the language of list comprehensions. The following algebraic data type defines its abstract syntax:

```
data LCexp = LC String Exp [LCQualifier]
           | LCAppend LCexp LCexp
           | LCAppl String LCexp
           | LCEnum Exp Exp
           | LCTabAccess String SubScripts
           | LCIf Exp LCexp LCexp
           | LCEmpty
                deriving (Show, Eq)

data LCQualifier = Filter Exp
                 | Generator Exp LCexp
                 | Let Exp Exp
                deriving (Show, Eq)

type LCProd = (String, LCexp)
```

The main construct of a list comprehension is element LC. It contains an expression (Exp, see Section 3.4) as head of the list comprehension and a list of qualifiers (*LCQualifier*) as the body. A qualifier then is either a filter, a generator or a let expression. *LCAppend* appends two list comprehensions. This is used for applications of the $||_{\mathcal{S}}$ combinator. An application of a choice function is mapped to a *LCAppl* construct, which applies a function to a list comprehension. *LCEnum* represents an enumeration of the form $[a\,..\,b]$. Nonterminals are mapped to a *LCTabAccess* construct. *LCIf* is used for filter applications and finally, *LCEmpty* represents the empty list.

4.4.2 Terminal parsers

Terminal symbol are represented as list comprehensions. See also the terminal parser definitions in Section 2.5.1.

```
terminals :: [(String, (Int, [Exp] → LCexp))]
terminals =
  [("empty", (0, λ[ i, j] → LC "empty" j [])),
   ("achar", (0, λ[ i, j] → LC "achar" (ExpFA "getInput" j) [])),
   ("astring", (0, λ[ i, j] → LC "astring" (ExpTupel [ i, j]) [])),
   ("astringp", (0, λ[ i, j] → LC "astringp" (ExpTupel [ i, j]) [])),
   ("char", (1, λ[ExpChar c, i, j] →
                LC "char" (ExpFA "getInput" j)
                  [Filter (ExpBinOp (ExpFA "getInput" j) "==" (ExpChar c))]))]
   ("string", (1, λ[ExpString s, i, j] →
                   LC "string" (ExpFA "getInputs" (i, j))
                     [Filter (ExpBinOp (ExpFA "getInputs" (i, j)) "==" (ExpString s))])),
   ("loc", (0, λ[ i, j] → LC "loc" j [])),
```

Overview: Translating into language of list comprehension
The function *sToLC* translates the intermediate language $\mathcal{S}$ into the language of list comprehensions, $\mathcal{LC}$.

sToLC : $\mathcal{S} \rightarrow \mathcal{LC}$ transformation
sToLC $u_p \rightarrow lc_p$

Input	*Output*
u_p $\mathcal{S}$-production	lc_p: list comprehension expression

Figure 4.11: $\mathcal{S} \rightarrow \mathcal{LC}$ transformation

4.4.3 $\mathcal{S} \rightarrow \mathcal{LC}$ translation

We implement the transformation $\mathcal{S} \rightarrow \mathcal{LC}$ by structural recursion. Figure 4.11 gives an overview.

The first definition is that of the terminal symbol $Terminal_{\mathcal{S}}$. The function *lookupTerminal* fetches the corresponding definition out of the list *terminals* defined above. We then apply the index pair (i, j) and the additional arguments on the terminal's definition.

```
sToLC :: ILUnit → LCexp
sToLC (Terminal_S (name, args) (i, j)) = lookupTerminal name args [ i, j]
```

$Nonterminal_{\mathcal{S}}$ is directly mapped on *LCTabAccess* of language $\mathcal{LC}$:

$$sToLC\ (Nonterminal_{\mathcal{S}}\ name\ (i,\ j))\ =\ LCTabAccess\ name\ (i,\ j)$$

An application of combinator $:|||_{\mathcal{S}}$ is mapped on *LCAppend*:

$$sToLC\ (p\ :|||_{\mathcal{S}}\ q)\ =\ LCAppend\ (sToLC\ p)\ (sToLC\ q)$$

We use the function *getAlgebraDef* to fetch the definition of the choice function and translate this definition to *LCAppl*:

$$sToLC\ (p\ \ :\ldots_{\mathcal{S}}\ h)\ =\ LCAppl\ (getAlgebraDef\ h)\ (sToLC\ p)$$

When processing filter applications, we distinguish between two kinds of filters: The length-limiting filters *size*, *minsize* and *maxsize* and all other filters. In this translation we do not need to handle the length-limiting filters, since we have already incorporated them during the subscript-calculation. All other filters are directly transformed into an *LCIf*-expression.

$$\begin{array}{l} sToLC\ (p\ \text{`With`}_{\mathcal{S}}\ (name,\ (i,\ j)))\ = \\ \quad \textbf{if}\ elem\ name\ [\texttt{"size"},\ \texttt{"minsize"},\ \texttt{"maxsize"}]\ \textbf{then}\ sToLC\ p \\ \quad \textbf{else}\ LCIf\ (lookupFilter\ name\ [\,i,\ j\,])\ (sToLC\ p)\ LCEmpty \end{array}$$

The application of the $:<<<_{\mathcal{S}}$ combinator is the central equation of the *sToLC* transformation. This is directly compiled into a list comprehension. The function *getAlgebraDef* fetches the definition for the algebra function f. We store the arguments of this algebra function in the list *arguments_f*, and the definition of the algebra function in the variable *rhs*. The expressions that are to be applied to this algebra function are converted into a list by the function *flat*. These are then translated by structural recursion on function *sToLC* and assigned to the corresponding arguments of the function's definition (*zip arguments_f argument*). The yield size check stored in the variable *bounds* is directly mapped to a filter application in the list comprehension. Loops and variable bindings are translated to generators of the list comprehension.

$$\begin{array}{l} sToLC\ ((f,\ bounds,\ loops)\ :<<<_{\mathcal{S}}\ p)\ =\ LC\ ""\ rhs\ (Filter\ bounds +\!\!+ \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad map\ Generator\ loops +\!\!+ \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad map\ Generator\ binding) \\ \textbf{where} \\ \quad (arguments\_f,\ rhs)\ =\ getAlgebraDef\ algfs\ f \\ \\ \quad arguments\ =\ map\ (sToLC)\ (flat\ p) \\ \quad flat\ (p\ :\sim\sim\sim_{\mathcal{S}}\ q)\ =\ flat\ p +\!\!+ flat\ q \\ \quad flat\ p\ =\ [p] \\ \quad binding\ =\ zip\ arguments\_f\ arguments \end{array}$$

As example we use the ADP-program for a global alignment:

```
alignment = tabulated (
                nil <<< (char '$') |||
                d <<< achar  −∼∼ alignment |||
                i <<< alignment ∼∼− achar |||
                r <<< achar  −∼∼ alignment ∼∼− achar ... h);

nil _ = 0
d a s = s − 1
i s a = s − 1
r a s b = if a ≡ b then s + 1 else s − 1
```

The subscript analysis generates the following $\mathcal{S}$-program:

```
alignment ! (i, j) =
  h [
    if (j − i) ≡ 1 then {
      nil ((char '$') (i, j))
    }
    ++
    if (j − i) ≥ 2 then {
      d (achar (i, i + 1), alignment ! (i + 1, j))
    }
    ++
    if (j − i) ≥ 2 then {
      i (alignment ! (i, j − 1), achar (j − 1, j))
    }
    ++
    if (j − i) ≥ 3 then {
      r (achar (i, i + 1), alignment ! (i + 1, j − 1), achar (j − 1, j))
    }
  ]
```

The transformation $\mathcal{S} \rightarrow \mathcal{LC}$ then yields the following list comprehension:

```
alignment =
  maximum (
    [0 | (j − i) ≡ 1, a1 ← [z ! j | z ! j ≡ '$']]++
    [a2 − 1 | (j − i) ≥ 2, a1 ← [z ! (i + 1)],
                           a2 ← alignment ! (i + 1, j)]++
    [a1 − 1 | (j − i) ≥ 2, a1 ← alignment ! (i, j − 1),
                           a2 ← [z ! j]]++
    [if a1 ≡ a3 then a2 + 1 else a2 − 1 |
```

$$(j-i) \geq 3,\ a1 \leftarrow [z\,!\,(i+1)],\ a2 \leftarrow alignment\,!\,(i+1,\ j-1),\ a3 \leftarrow [z\,!\,j]])$$

4.4.4 List elimination

In the above example it is remarkable, that all list comprehensions can only contain atomic elements. For example, the generator $a1 \leftarrow [z\,!\,(i+1)]$ is in any case only singleton. The table *alignment* can also contain only maximally one element, since it closes with the choice function *maximum*. Therefore the generator $a2 \leftarrow alignment\,!\,(i+1,\ j)$ can also hold maximally one element. With this information we remove superfluous generators from the list comprehensions. We achieve this by moving the corresponding generators directly into the heads of the list comprehensions. This optimization then leads to the following $\mathcal{LC}$-program:

```
alignment =
  maximum (
    [0 | (j - i) ≡ 1, z ! j ≡ '$']++
    [tbl_alignment (i + 1, j) - 1 | (j - i) ≥ 2]++
    [tbl_alignment (i, j - 1) - 1 | (j - i) ≥ 2]++
    [if z ! (i + 1) ≡ z ! j then tbl_alignment (i + 1, j - 1) + 1
                            else tbl_alignment (i + 1, j - 1) - 1 |
                                 (j - i) ≥ 3])
```

In the following, we will translate this intermediate code into the target program.

4.4.5 Target code generation

The following algebraic data type defines the abstract syntax of the target language:

```
data TL = TLVar Ident
        | TLAssign Ident TL
        | TLIf Exp [TL] [TL]
        | TLComment String
        | TLFor Ident Exp Exp [TL]
        | TLNonterm Ident Subscripts
        | TLExpr Exp
              deriving (Show, Eq)
```

Here, *TLVar* holds a variable in the target language. *TLAssign* is an assignment of a target language construct to a variable. The *TLIf* constructs represents an if-Expression.

TLComment is a comment in the target language. *TLFor* is a for-loop, *TLNonterm* a use of a nonterminal symbol. This is then either mapped to a table access (in case of a tabulated nonterminal) or a function call (in case of a nontabulated nonterminal). Finally, *TLExpr* can contain an expression in form of the data type *Exp*.

Overview: Code generation
The function *cg* translates the language of list comprehensions, $\mathcal{LC}$, into the target language $\mathcal{TL}$.

cg : code generation
cg $lc_p \rightarrow (v_{result},\ tc_p)$

Input	*Output*
lc_p: list comprehension expression	v_{result}: result variable
	tc_p: target code

States
temporary variables
loop variables
current choice function

Figure 4.12: Code generation

The function cg (Figure 4.12) is implemented via a state monad. The state consists of the following elements:

- A list of temporary variables. With the function *newTempVar* we can fetch a new temporary variable from the store of variables. The store itself is unlimited.
- The list of loop variables. With the function *addLoopVar* we can add an additional loop variable to the store of variables.
- The current choice function. This function can be changed with function *changeChoice* and accessed with function *getChoice*.

We begin our presentation with element *LCAppl*, the application of a choice function. Here the current function is simply changed with the state monad function *changeChoice*:

```
cg (LCAppl chc lc) =
 do
   changeChoice chc
   cg lc
```

With the application of a concatenation, the two expressions are first translated by structural recursion and the results are stored in the variables *v1* and *v2*. Then a new temporary variable *v3* is introduced for storing the overall result of the concatenation. The code for the actual selection is generated by the function *cgChoiceSingleElement*.

```
cg (LCAppend lc1 lc2) =
 do
   (v1, code1) ← cg lc1
   (v2, code2) ← cg lc2
   v3 ← newTempVar TLInt
   chc ← getChoice
   chcCode ← return (cgChoiceSingleElement v3 chc v1 v2)
   return (v3, code1 ++ code2 ++ chcCode)
```

The compilation of an If-construct behaves similarly. The variables *v1* and *v2* hold the results for the then- and the else-case. The variable *v3* then stores the overall result. The If-construct is directly compiled to an If-construct of the target language:

```
cg (LCIf exp lc1 lc2) =
 do
   (v1, code1) ← cg lc1
   (v2, code2) ← cg lc2
   v3 ← newTempVar TLInt
   code ← return [TLIf exp (code1 ++ [TLAssign v3 (TLVar v1)])
                           (code2 ++ [TLAssign v3 (TLVar v2)])]
   return (v3, code)
```

A table access in the $\mathcal{LC}$-language is compiled directly to a table access in the $\mathcal{TL}$ language:

```
cg (LCTabAccess nt ss) =
 do
   v1 ← newTempVar TLInt
   code ← return [TLAssign v1 (TLNonterm nt ss)]
   return (v1, code)
```

For the empty list an assignment of the zero value is produced (function *nilval*). The zero value depends on the choice function. With maximization the zero value is INTMIN, with minimization it is INTMAX, and with summation it is 0.

```
cg LCEmpty =
 do
   v1 ← newTempVar TLInt
```

```
    chc ← getChoice
    code ← return [TLAssign v1 (nilval chc)]
    return (v1, code)
```

The largest part of the translation takes place during the actual list comprehension. Here, the additional function *cgLC* serves as translator for the qualifiers of the list comprehension.

```
cg (LC cmt exp lcs) =
 do
    comment ← return [TLComment [cmt]]
    (v, code) ← cgLC [] exp lcs
    return (v, comment ++ code)
```

The function *cgLC* has three arguments. The variable *bind* stores the bindings of the algebra function's variables to those of the corresponding qualifiers. The variable *exp* stores the definition of the algebra function. The third argument is the list of qualifiers for the list comprehension.

A filter-qualifier is translated into an If-construct of the target language:

```
cgLC bind exp ((Filter filterExp) : lcs) =
 do
    (v1, code1) ← cgLC bind exp lcs
    chc ← getChoice
    code2 ← return [TLIf filterExp code1 [TLAssign v1 (nilval chc)]]
    return (v1, code2)
```

An enumeration generator is translated into a loop of the target language:

```
cgLC bind exp ((Generator (ExpVar k) (LCEnum (from) (to))) : lcs) =
 do
    (v1, code1) ← cgLC bind exp lcs
    addLoopVar k
    v2 ← newTempVar TLInt
    chc ← getChoice
    code2 ← return [TLAssign v2 (nilval chc),
                    TLFor k from to (code1 ++ cgChoiceSingleElement v2 chc v2 v1)]
    return (v2, code2)
```

All other generators are recursively compiled via *cg lcexp* and the result variable of this compilation (*v1*) is bound to the corresponding generator-variable (*a*).

```
cgLC bind exp ((Generator (ExpVar a) lcexp) : lcs) =
```

```
  do
    (v1, code1) ← cg lcexp
    bind2 ← return [(a, v1)]
    (v2, code2) ← cgLC (bind ++ bind2) exp lcs
    return (v2, code1 ++ code2)
```

If the list of qualifiers is empty, the actual body of the list comprehension can be produced. The function *insertVarbinds* replaces all occurrences of the qualifier variables by the corresponding variables of the target language:

```
cgLC bind exp [] =
  do
    v ← newTempVar TLInt
    code ← return [TLAssign v (TLExp (insertVarBinds bind exp))]
    return (v, code)
```

For our example, the function *cg* generates to following target code:

```
static void calc_alignment(int i, int j)
{
   int v1, v2, v3, v4, v5, v6, v7;

   /* nil <<< char '$' */
   if ((j-i) == 1) {
      if (z[j] == '$') {
         v1 = 0;
      }
      else {
         v1 = INTMIN;
      }
   }
   else {
      v1 = INTMIN;
   }
   /* d <<< achar ~~~ alignment */
   if ((j-i) >= 2) {
      v2 = tbl_alignment(i+1, j) - 1;
   }
   else {
      v2 = INTMIN;
   }
   v3 = v1 > v2 ? v1 : v2;
   /* i <<< alignment ~~~ achar */
   if ((j-i) >= 2) {
      v4 = tbl_alignment(i, j-1) - 1;
```

```
   }
   else {
      v4 = INTMIN;
   }
   v5 = v3 > v4 ? v3 : v4;
   /* r <<< achar ~~~ alignment ~~~ achar */
   if ((j-i) >= 3) {
      v6 = (z[i+1] == z[j]) ?
         tbl_alignment(i+1, j-1) + 1 :
         tbl_alignment(i+1, j-1) - 1 ;
   }
   else {
      v6 = INTMIN;
   }
   v7 = v5 > v6 ? v5 : v6;
   tbl_alignment(i, j) = v7;
}
```

4.5 Table design

This section is taken from [50].

Finding good and optimal table configurations is the problem studied in this section. Previously, this has been the programmer's responsibility – explicitly so when using the ADP approach, or implicitly otherwise. A most ambitious goal would be to completely automate this step, virtually freeing the program designer from efficiency concerns. However, our main result here shows that a complete automation of table design is computationally infeasible: Both the choice of a good table configuration and the choice of an optimal configuration are NP-complete problems. Consequently, we develop a pragmatic approach that tries to reduce the problem size by various means to the point where the problem can be solved to optimality.

4.5.1 Running Example: Palindromic structures in strings

As a running example, we shall use the analysis of palindromic structures in strings. A *separated* palindrome [21] is a string of the form $uv(u^{-1})$. In "abcdeba", for example, the separator v could be chosen to be "cde", "bcdeb", or "abcdeba". Intuitively, we might say that the first choice is the best, as it maximizes the length of u. Generally, we have some scoring scheme that determines which palindromic structure is best for a given string. In *approximate* separate palindromes we allow differences between u and u^{-1}, using the familiar string edit model with replacements, deletions and insertions. With such generalization, the

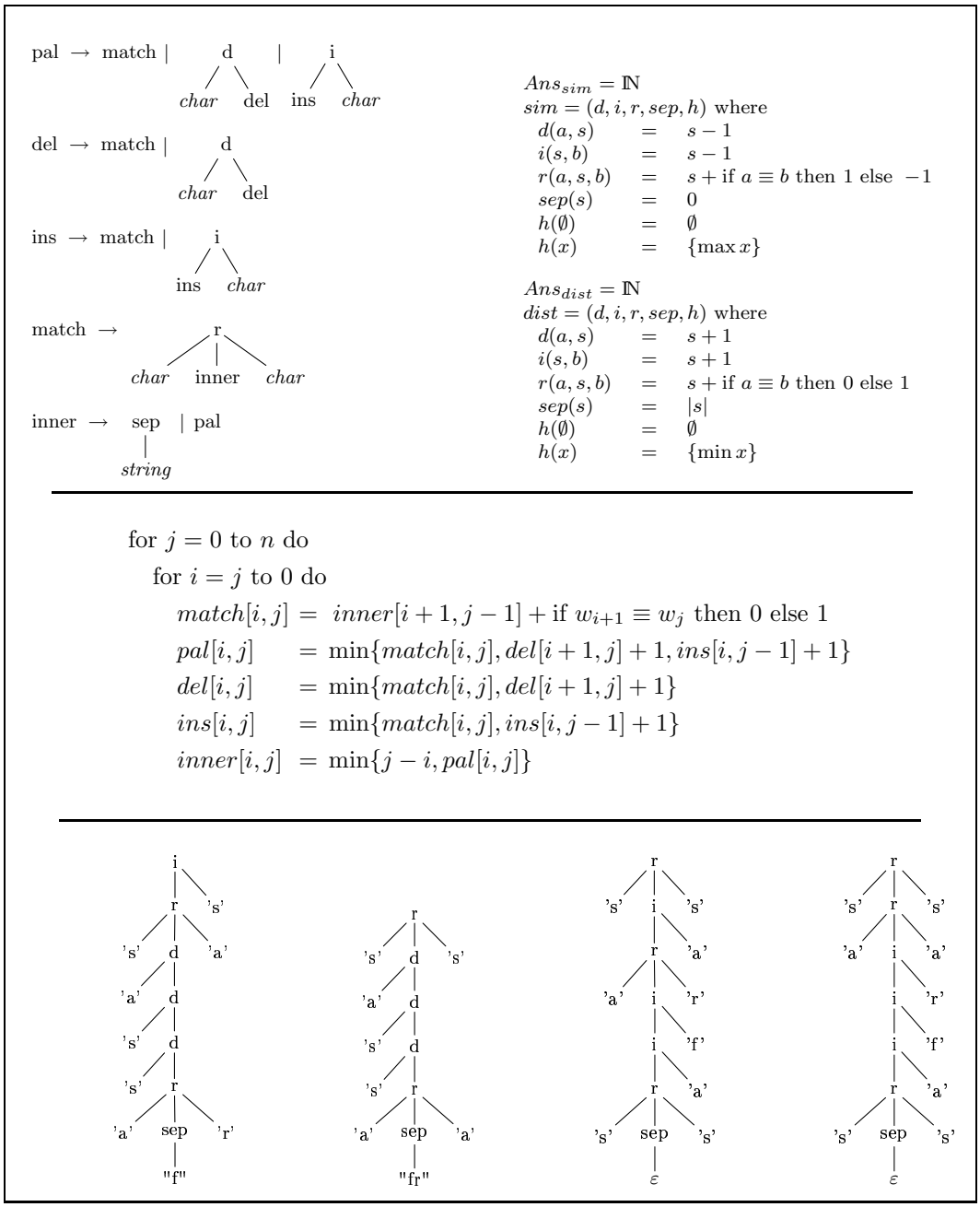


Figure 4.13: Top: Example yield grammar and two evaluation algebras. The axiom symbol is *pal*, and *char* denotes an arbitrary character. Middle: The corresponding recurrences specialized for algebra *dist*. For lack of space, the initialization equations are omitted. Bottom: Four candidate structures in the search space for the best approximate separated palindrome structure for the string "sassafras". Under algebra *dist* the candidates evaluate to scores (from left to right) 7, 5, 4 and 3.

number of alternative palindromic structures for a given string becomes very large. Using different scoring schemes, we may formulate the optimization objectives of maximizing self-similarity, or minimizing differences, with slightly different applications.

Figure 4.13 collects our formulation of the approximate separated palindrome example in the ADP framework. It shows (top left) the yield grammar defining approximate, separated palindromic structures. Recall that any string can be parsed as such a palindrome in many ways, with the evaluation algebra determining the best palindromic structure. Four candidates for the string "sassafras" are shown in the bottom part of the figure. Two evaluation algebras are given (top right), algebra *sim* maximizing self-similarity, and algebra *dist* minimizing differences. The middle part of Fig. 4.13 shows the explicit recurrences derived from the grammar and algebra *dist*.

4.5.2 Optimal tabulation

In this section we describe how to determine the nonterminals whose results need to be stored in dynamic programming tables. We begin with the best possible runtime for a given yield parser.

When *all* nonterminal symbols are tabulated, the runtime efficiency of the tabulating yield parser is known. It depends on the *width* of the grammar:

Definition 4 *(Width of productions and grammar) Let t be a tree pattern, and let k be the number of nonterminal or lexical symbols in t whose yield size is not bounded from above. We define $width(t) = k-1$. Let $P_v = \{t | v \to t \in P\}$. We define $width(v) = \max_{t \in P_v}\{width(t)\}$ and $width(\mathcal{G}) = \max_{v \in V}\{width(v)\}$.*

In this case, the execution time is $O(n^{2+width(\mathcal{G})})$ [17]. This implies minimal runtime, but maximal space consumption, and is what we want to improve upon by a more clever table design.

In Fig. 4.13, middle part, we show the recurrences that implement the tabulating yield parser for the example grammar. Efficiency analysis in this case is simple. With all intermediate results stored in the five tables $match, \ldots, inner$, the runtime is solely determined by the outer for-loops which leads to an execution time of $O(n^2)$. In case of productions with $width(v) \geq 1$, for example $v \to N(string, v)$, the recurrence equations and the eventual implementation of the corresponding parsers would contain additional loops for moving subword boundaries resulting in optimal runtimes of $O(n^3)$ or more. While tabulating everything yields optimal efficiency in terms of execution time, it may require more space than really needed. Conversely, the yield parser may implement a parser for each nonterminal symbol as a recursive function – resulting in low space requirements for tables (none, to be precise), but exorbitant inefficiency, and most likely a runtime stack overflow. The solution is to assign a table configuration to the grammar that achieves an optimal balance.

Efficiency analysis for a given table configuration

In order to make more precise statements about the usage of tables and recursive functions, we define:

Definition 5 *(Table configuration) Let $\mathcal{G} = (V, Z, P)$ be a tree grammar over Σ and $P_{\mathcal{G}}$ the corresponding yield parser. A* table configuration *ϑ is a subset $\vartheta \subseteq V$ denoting that for all $q \in \vartheta$ the parser p_q is to be tabulated and for all $q \notin \vartheta$ the parser p_q is to be implemented as a recursive function. $P_{\mathcal{G}}^{\vartheta}$ shall represent the yield parser $P_{\mathcal{G}}$ for yield grammar $(\mathcal{G}, y)$ under table configuration ϑ. $\mathcal{P}(V)$ denotes the set of all possible table configurations.*

Note that the semantics of the tabulating yield parsers $P_{\mathcal{G}}^{\vartheta}$ are equivalent for all $\vartheta \in \mathcal{P}(V)$. We will distinguish between *good* and *optimal* table configurations.

Definition 6 *(Good and optimal table configurations) A configuration is* good *if it uses the minimal number of tables that leads to polynomial runtime. A configuration is* optimal *if it uses the minimal number of tables that leads to a runtime with the best possible polynomial degree.*

Note that our notion of optimality includes a space-time trade-off. A grammar with a good table configuration may still contain a subgrammar whose dependencies raise runtime complexity to an arbitrary (but fixed) polynomial degree. Extra tables can be assigned to reduce this degree – hence an optimal configuration, in general, holds more tables than a good one. For example, the second configuration shown in Fig. 4.14 is good (using 1 table), while the first configuration is optimal (using 3 tables).

Definition 7 *(Dependence mapping) Let $S = V \cup L$ be the set of nonterminal and lexical symbols in $\mathcal{G}$. The* dependence mapping *u for $q \in V$ and input length n is given by $u(q, n) = (d_1, \ldots, d_r)$ where each $d_k, 1 \leq k \leq r$ represents a dependence property $d_k \in S \times \mathbb{N}$ such that $d_k = (q_k, i_{q_k} + j_{q_k})$ if and only if the parser $p_q(0, n)$ uses the result of $p_{q_k}(i_{q_k}, n - j_{q_k})$. For convenience, we simply denote $(q_k, s_{q_k}) \in u(q, n)$.*

Theorem 2 *The runtime of an ADP algorithm implemented by a tabulating yield parser $P_{\mathcal{G}}^{\vartheta}$ on input w of length n is given by $r(P_{\mathcal{G}}^{\vartheta}, n)$ where:*

$$\bar{r}(\vartheta, q, 0) = 1 \tag{4.6}$$

$$\bar{r}(\vartheta, q, n) = \sum_{(q', s_{q'}) \in u(q,n)} \begin{cases} 1 & \text{for } q' \in L \\ 1 & \text{for } q' \in V, q' \in \vartheta \\ \bar{r}(\vartheta, q', n - s_{q'}) & \text{otherwise} \end{cases} \tag{4.7}$$

$$r(\vartheta, q, n) = \begin{cases} n^2 \cdot \bar{r}(\vartheta, q, n) & \text{for } q \in \vartheta \\ \bar{r}(\vartheta, q, n) & \text{otherwise} \end{cases} \tag{4.8}$$

$$r(P_{\mathcal{G}}^{\vartheta}, n) = O(\max_{q \in V} r(\vartheta, q, n)) \tag{4.9}$$

Proof. The dependence mapping $u(q, n)$ provides all calls to lexical or nonterminal parsers in the calculation of parser p_q on input length n. Without loss of generality we assume that each parser terminates at an input of length 0 (4.6). Lexical parsers and table accesses (4.7) can be performed in constant time. In case of a non-tabulated parser $p_{q'}$ (4.7), the parser $p_{q'}$ is called with an input length reduced by $s_{q'}$. The outer for-loops for a tabulated parser lead to a minimal effort of at least $O(n^2)$ (4.8) and the overall runtime of the algorithm (4.9) is determined by the parser with the maximal asymptotic runtime. □

This of course gives sparse information about a closed form for $r(P^{\vartheta}_{\mathcal{G}}, n)$. All we know is that if all productions are tabulated, the runtime is solely affected by the constants in (4.7) and the outer for-loops of (4.8). In case of non-tabulated productions, (4.7) is defined recursively, which makes reasoning about complexity more difficult. Tabulating no production at all will in most cases lead to an exponential runtime. In the following we will investigate how to find the minimal numbers of tables needed to achieve a polynomially bounded runtime $r(P^{\vartheta}_{\mathcal{G}}, n)$.

Staying polynomial

The number of dependences for a parser p_q is bounded by $|u(q, n)| = O(n^{width(q)})$. Therefore, $r(P^{\vartheta}_{\mathcal{G}}, n)$ can only become exponential in the presence of recursive calls between parser functions (Eq. 4.7).

For the following developments it is convenient to introduce some terminology for graphs: A directed graph $G = (V, E)$ is given by a nonempty set $V = \{v_1, \ldots, v_s\}$ of vertices and a set $E = \{e_1, \ldots, e_m\}$ of edges consisting of ordered pairs of elements of V. A directed graph with multiple edges is called a directed *multigraph* with edges given as a multiset E. A *loop* is an edge connecting a vertex with itself. In the following we only consider directed multigraphs with loops allowed which we will simply call graphs. In a *weighted graph* each edge e_i is also associated with a weight $w(e_i)$.

A directed *walk* is a sequence $\pi = v_0 e_1 v_1 e_2 \ldots e_l v_l$, where $v_i \in V, e_j \in E$ for $0 \leq i \leq l, 1 \leq j \leq l$, and each edge e_j is directed out of vertex v_{j-1} and directed into vertex v_j. Note that the edges and vertices in π are not necessarily distinct. The length of a walk is the number of edges in the sequence and is denoted by $l(\pi)$. The weight of a walk is the sum of the weights of its edges and is denoted by $\tilde{w}(\pi) = \sum_{i=1}^{l(\pi)} w(e_i)$.

A *circuit* is a closed walk with $v_0 = v_l$ and all edges distinct. Under this definition a loop is also a circuit. Two circuits are distinct if one is not a cyclic permutation of the other. The *circuit degree* $\delta(v)$ of a vertex v shall be the number of distinct circuits containing v. We define the circuit degree of a graph G as $\Delta(G) = \max_{v \in V} \delta(v)$. A circuit which contains no vertex more than once (apart from the initial one), is called *elementary*.

The dependence mapping given by u for a yield parser $P^{\vartheta}_{\mathcal{G}}$ can be represented by a weighted directed *dependence graph* G with the nonterminal set V as vertex set and an edge from q to q_k with weight s_{q_k} for each $(q_k, s_{q_k}) \in u(q, n)$. Such a graph represents all dependences,

regardless whether by table access or by function call. A graph limited to function call dependences can be derived by restricting the edge set to edges directed into vertices from $V \backslash \vartheta$.

In the following we restrict the analysis to grammars $\mathcal{G}$ with $width(\mathcal{G}) = 0$. This leads to parsers with constant dependence, i.e. all references to parser functions have the form $p_q(i + i_q, j - j_q)$, with $i_q, j_q \in \mathbb{N}$ and $i_q, j_q \geq 0$. This restriction is necessary in order to obtain graphs that are independent from the input length n contained in the dependence mapping u.

Definition 8 *(Dependence graph) Let $P^{\vartheta}_{\mathcal{G}}$ be a yield parser for yield grammar $(\mathcal{G}, y)$ under table configuration ϑ and let u be the corresponding dependence mapping. The* dependence graph $G(P^{\vartheta}_{\mathcal{G}})$ *is given by the grammar's nonterminal set V as vertex set and edges $E = \{e_1, \ldots, e_m\}$. Edge $e_i = (q, q_k)$ with $w(e_i) = s_{q_k}$ is in E if and only if $q_k \notin \vartheta$ and $(q_k, s_{q_k}) \in u(q, n)$. For convenience, we denote G for $G(P^{\vartheta}_{\mathcal{G}})$.*

The use of the dependence graph is influenced by [28] where such graphs are used to represent systems of uniform recurrence equations. Figure 4.14 shows three graphs with different table configurations and circuit degrees $\Delta(G)$ for the palindrome example.

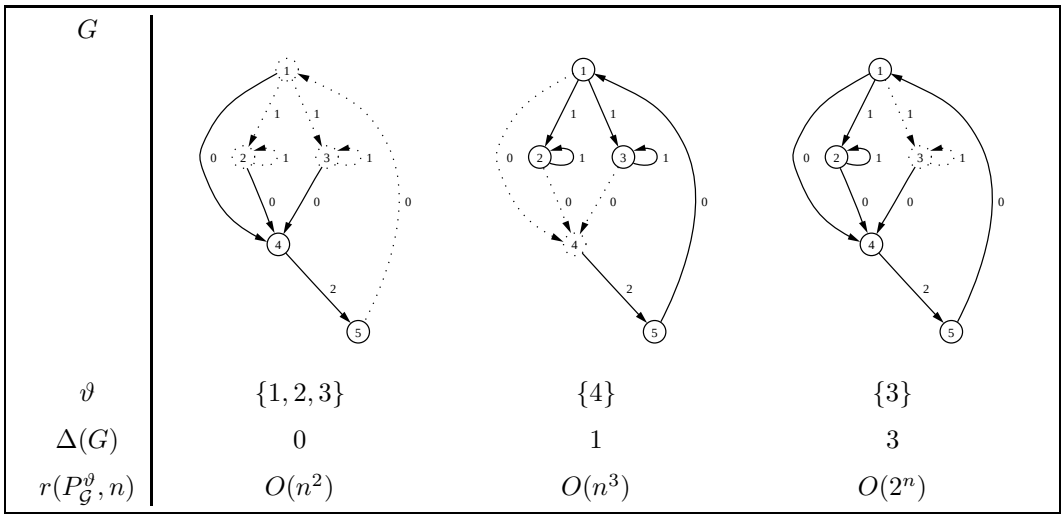


Figure 4.14: Three dependence graphs for the yield grammar of Example 4.5.1, with corresponding table configurations ϑ and circuit degrees $\Delta(G)$. The vertices are numbered as follows: $1 \rightarrow$ `pal`, $2 \rightarrow$ `del`, $3 \rightarrow$ `ins`, $4 \rightarrow$ `match`, $5 \rightarrow$ `inner`. Table access dependences are drawn as dotted edges. The runtime of the yield parser $P_{\mathcal{G}}$ under table configuration ϑ is denoted by $r(P^{\vartheta}_{\mathcal{G}}, n)$.

Each edge in the dependence graph represents a function call between two parser functions.

A walk π in the graph can therefore be interpreted as a sequence of consecutive function calls where the weight $\tilde{w}(\pi)$ of the walk is the length of the "consumed" input and the length $l(\pi)$ is the number of required function calls.

This leads to the following relation between the runtime of a yield parser and its dependence graph:

Theorem 3 *Let $P_{\mathcal{G}}^{\vartheta}$ be a tabulating yield parser with corresponding dependence graph G. Then, $P_{\mathcal{G}}^{\vartheta}$ has polynomial runtime if and only if $\Delta(G) \leq 1$.*

Proof. If there is no vertex on at least two different circuits, each recursive function is called at most once for each combination of parameters, such that the runtime is polynomial. If there is such a vertex v, a call on v can trigger two recursive calls to v. Since the recursion depth is linear, this leads to $O(2^n)$ calls, and the runtime is exponential. □

A more detailed proof of Theorem 3 can be found in Appendix A.

NP-completeness of table design

Theorem 3 gives a convenient property to distinguish between polynomial and exponential runtimes of a tabulating yield parser. In order to find good table configurations, we need to search for configurations that lead to dependence graphs with $\Delta(G) \leq 1$ while using a minimal number of tables.

A simple algorithm is to systematically test all possible table configurations for their impact on the number of circuits in the dependence graph. Such approach would be exponential in the number of nonterminal symbols, so the question arises whether a better algorithm exists.

Graph theory is a stomping ground for NP-complete and harder problems and provides a rich theory on these kinds of questions. And indeed, we can utilize a powerful result from graph theory to give an answer to the question stated above:

Theorem 4 *(NP-completeness of good table design) Finding the minimal number of tables to prevent exponential runtime of an ADP algorithm implemented by a tabulating yield parser $P_{\mathcal{G}}$ is NP-complete.*

Proof. Recall the construction of the dependence graph in Definition 8. The edge set E for the minimal table configuration $\vartheta = \emptyset$ represents all dependences between parsers for the situation that no results are tabulated. Adding a table for the parser p_q to the configuration results in deletion of all edges directed into the corresponding vertex q. Obviously, for the number of circuits in the graph this has the same effect as deleting the vertex q itself. This leads to the observation that finding the minimal number of tables to prevent exponential runtime is equivalent to finding the minimal number of vertices which must be deleted from

the dependence graph such that the resulting graph contains no vertex which is part of two or more circuits. And this is exactly a classic node-deletion problem:

For a fixed graph property Π*, find the minimum number of nodes (or vertices) which must be deleted from a given graph so that the result satisfies* Π. [29]

It was shown in [29] that if Π is a "nontrivial" property which is "hereditary" on induced subgraphs, then the node-deletion problem for Π is NP-hard. Furthermore, if testing for Π can be performed in polynomial time, then the node-deletion problem for Π is NP-complete.

In our application, the property Π is: the graph contains no vertex which is part of two or more circuits. To finish the proof, we show that Π satisfies the properties claimed above. A graph property is *nontrivial* if infinitely many graphs satisfy it and infinitely many graphs fail to satisfy it. This is clearly given for Π. A property is *hereditary* if for a given graph satisfying Π, every node-induced subgraph also satisfies Π. It is obvious that all subgraphs of a graph containing no vertex being part of two or more circuits retain this property. Finally, Π can be tested for in polynomial time. A simple algorithm is to calculate all strongly connected components of the graph – which is linear in the number of vertices and edges [52] – and to check whether each of them is either a single vertex or an elementary circuit – which is linear as well. Hence, finding the minimal number of tables to prevent exponential runtime is NP-complete. □

Of course, in practical applications we are mostly interested in the *optimal* configurations, not necessarily in the good ones. We can use the same approach to show the NP-completeness of the problem of finding optimal configurations.

Theorem 5 *(NP-completeness of optimal table design) Finding the minimal number of tables that are necessary to achieve the best possible asymptotic runtime of an ADP algorithm implemented by a tabulating yield parser* $P_{\mathcal{G}}$ *is NP-complete.*

Proof. The optimal execution time of a yield parser $P_{\mathcal{G}}$ with $width(\mathcal{G}) = 0$ is $O(n^2)$. This is clear, since when all parsers are tabulated, the dependence graph is empty. Circuits in the corresponding dependence graph, arising from non-tabulated productions, represent table configurations that lead to suboptimal runtimes of at least $O(n^3)$. Therefore, in order to obtain optimal runtime for the algorithm, the dependence graph must be made cycle free. Achieving this by the minimal number of tables is equivalent to the node-deletion problem for acyclicity of a directed graph. And this is, as expected, also NP-complete [29]. □

In the course of proving Theorems 4 and 5, we have restricted the yield grammars considered. Our results, however, pertain to general yield grammars as well, because (1) arbitrary yield grammars contain the restricted ones as a subclass, such that their table design problem is at least NP-hard. (2) Testing goodness or optimality of a given table configuration works as described earlier also in the general case, hence NP-completeness is retained.

A pragmatic implementation strategy

Motivated by the above NP-completeness results, we present a pragmatic approach to the table design problem. It uses annotations by the programmer, preprocessing analyses, and a brute force algorithm.

The programmer's annotation distinguishes between two types of nonterminal symbols: The ones that shall be tabulated, and the ones that shall be implemented as nontabulated recursive functions. We denote them ϑ^a and ϱ^a, respectively.

Similarly, we consider two additional sets of nonterminals, ϑ^p and ϱ^p, derived from preprocessing phases. We use different phases, depending on whether we search for good or for optimal configurations:

Good configurations. Productions having more than one self-reference need to be tabulated in any case: $\vartheta^p = \{v | v \in V, \Delta(G(P_{\mathcal{G}}^{V\setminus\{v\}})) > 1\}$. Conversely, nonterminals not contained in any circuit can by no means be responsible for exponential runtime: $\varrho^p = \{v | v \in V, \delta(v) = 0\}$ with $G = G(P_{\mathcal{G}}^{\vartheta^a \cup \vartheta^p})$.

Optimal configurations. We derive the best possible polynomial degree by calculating the runtime for a full table configuration: $r_{opt} = r(P_{\mathcal{G}}^{V}, n)$ (see Theorem 2). When not tabulated, even a single self-recursive production or a production containing an inner loop can be responsible for an additional runtime factor. We identify these candidates and mark them for tabulation: $\vartheta^p = \{v | v \in V, r(P_{\mathcal{G}}^{V\setminus\{v\}}, n) > r_{opt}\}$. Clearly, nonterminals with a constant runtime need no tabulation: $\varrho^p = \{v | v \in V, r(\vartheta^a \cup \vartheta^p, v, n) = O(1)\}$.

The remaining nonterminals are free for optimization by the brute force approach: $F = V\setminus\{\vartheta^a \cup \varrho^a \cup \vartheta^p \cup \varrho^p\}$. Let Ω be a predicate on table configurations that determines whether the configuration is relevant for optimization at all. Corresponding to Theorem 3, we use $\Omega(\vartheta) = \Delta(G(P_{\mathcal{G}}^{\vartheta})) \leq 1$ for good configurations and $\Omega(\vartheta) = r(P_{\mathcal{G}}^{\vartheta}, n) \equiv r_{opt}$ for the optimal ones.

Let $\vartheta^i = \vartheta^a \cup \vartheta^p$ be the initial configuration. Then, $\Theta = \{\vartheta \cup \vartheta^i | \vartheta \in \mathcal{P}(F), \Omega(\vartheta \cup \vartheta^i)\}$ describes the set of configurations satisfying the predicate and $\Theta_{min} = \{\vartheta | \vartheta \in \Theta, |\vartheta| = min_{c\in\Theta}|c|\}$ describes the corresponding minimal configurations. Our brute force algorithm actually enumerates these sets and tests for minimality.

Should this strategy not be effective (due to computational effort for Θ_{min}), one must restart with a more detailed annotation by the programmer.

In the implementation, we must also deal with inner loops arising from productions with $width(v) \geq 1$, which played no role in the NP completeness proofs. During the construction of the dependence graphs we represent dependences from inside inner loops by *two* edges instead of a single one. Since our testing criterion is $\Delta(G) \leq 1$, this ensures that all configurations with circuits containing an inner loop are rejected.

Practical experience

The table design problem studied here mathematically arises in practice only when the algorithmic task at hand has a certain minimal level of sophistication. To support our claim of practical relevance, we give a short report on such an application. Our largest ADP-program so far is *pknotsRG* [45, 39], which predicts RNA secondary structures including the so-called pseudoknots. Its time and space requirements are $O(n^4)$ and $O(n^2)$. It uses a yield grammar with 47 nonterminal symbols. The productions split up in a total of 140 alternatives, which indicates the complexity of the case analysis involved. Therefore, space does not permit to explain the algorithm here. We show the dependence graphs for three different table configurations in Figures 4.15 to 4.17. The original implementation (Fig. 4.15) – developed using manual table design – has a configuration of 17 tables. With moderate annotation, our strategy derived good configurations of 4 tables (Fig. 4.16) and optimal configurations of 12 tables (Fig. 4.17).

Comparing the hand-made to the optimal design, we found that while saving nearly 30% space, the runtime of the optimized implementation increased by 19% – the constant factor due to the additional recursive function calls. In many applications in the bioinformatics domain, including this one, available space is the limiting factor. In such cases, one is thankful to accept a small slowdown in exchange for the ability to handle larger input data.

4.6 Interface creation

To this point we have given a detailed description of the translation of ADP programs to an imperative language like C. What is still missing is the generation of a suitable interface. In the embedded Haskell version this is no problem. The embedded version is typically used with the Haskell interpreter hugs or with the Haskell compiler ghc. In case of hugs the interface is provided by the interface of the interpreter. Here the user can call every function of the program directly from the hugs interface. In case of ghc, the ADP programmer has to implement the user interface directly in Haskell. Since the ADP compiler does not understand full Haskell, both approaches are not suitable for automated interface generation.

So far, three published ADP programs were compiled with the help of the ADP compiler: pknotsRG, RNAshapes, and RNAhybrid. The interfaces for these programs were implemented by hand in C. Figure 4.18 shows the pknotsRG interface. The RNAshapes interface is described in detail in Appendix B.2. The implementation of such interfaces is time consuming and error-prone. In the following, we describe how to generate such interfaces automatically.

The first thing to note is that a typical command line interface consists of several standard components:

- Standard options like -h for help, -v for the current version, and -f for file input

Figure 4.15: Dependence graph for *pknotsRG* – manual table design (17 tables). This table configuration leads to time and space requirements of $O(n^4)$ and $O(n^2)$. The tabulated nonterminals are shown as nodes in dotted ovals; their calling arcs are also dotted.

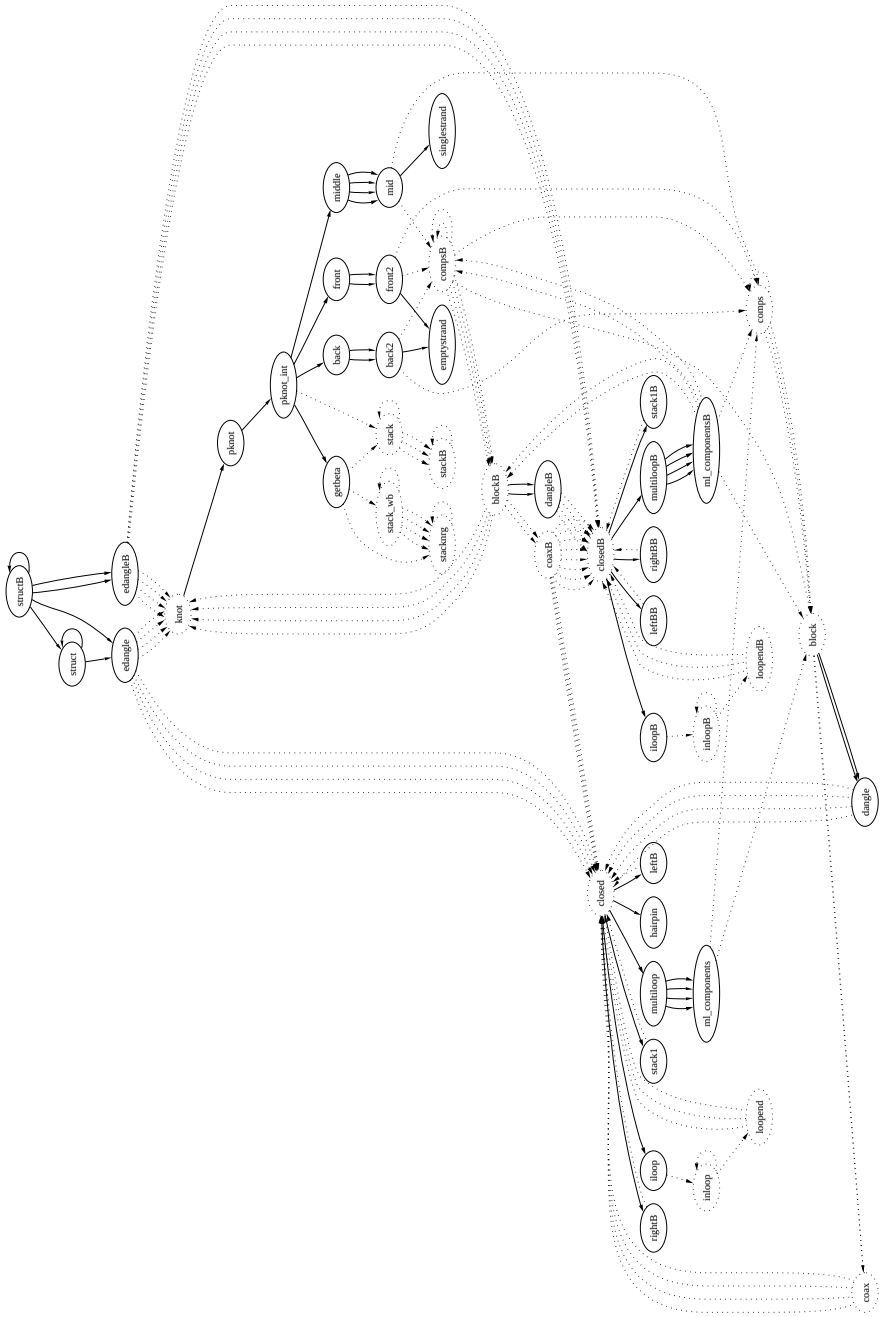


Figure 4.16: Dependence graph for *pknotsRG* – good table design (4 tables). This table configuration leads to time and space requirements of $O(n^7)$ and $O(n^2)$, respectively. Albeit polynomial, the runtime of $O(n^7)$ is surely unacceptable for practical usage.

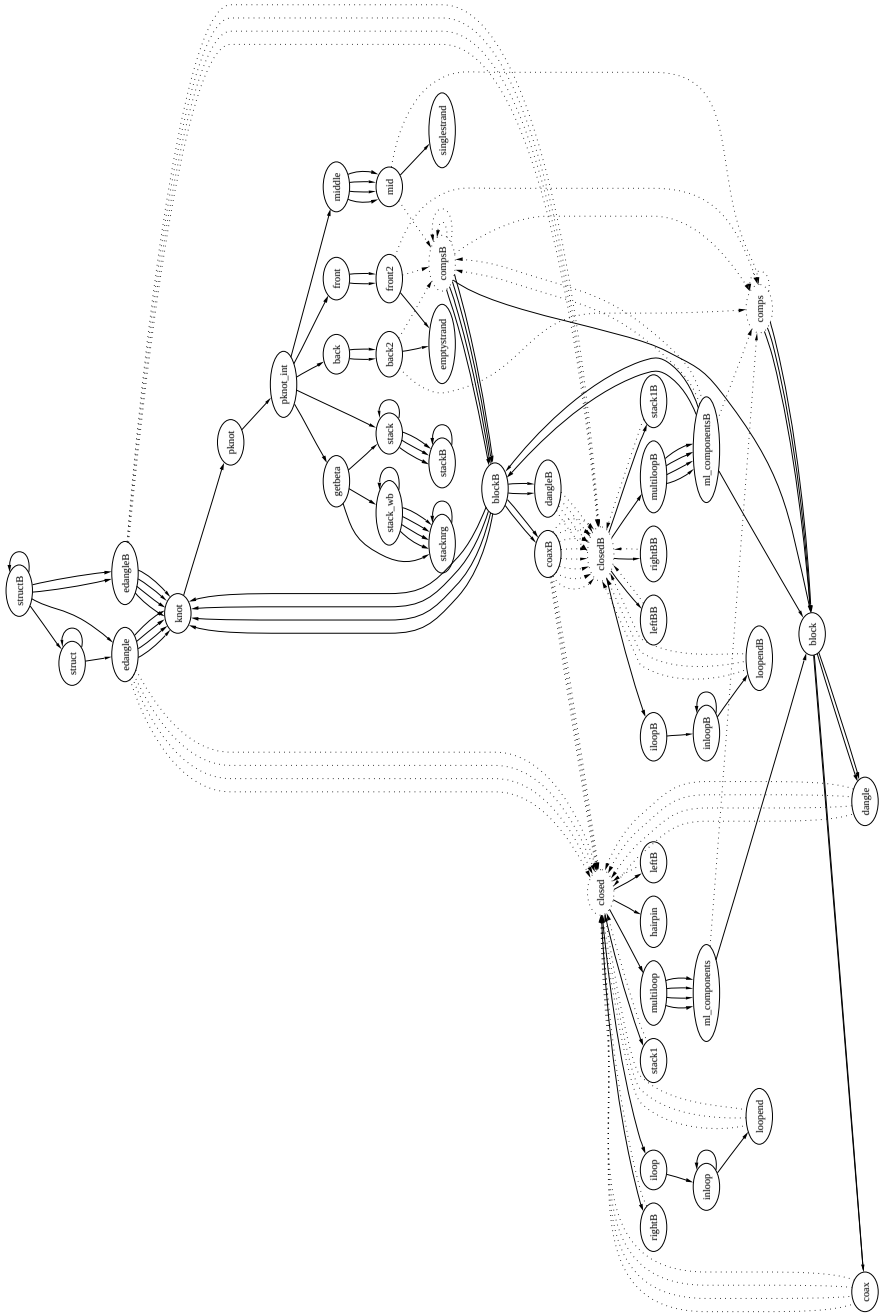
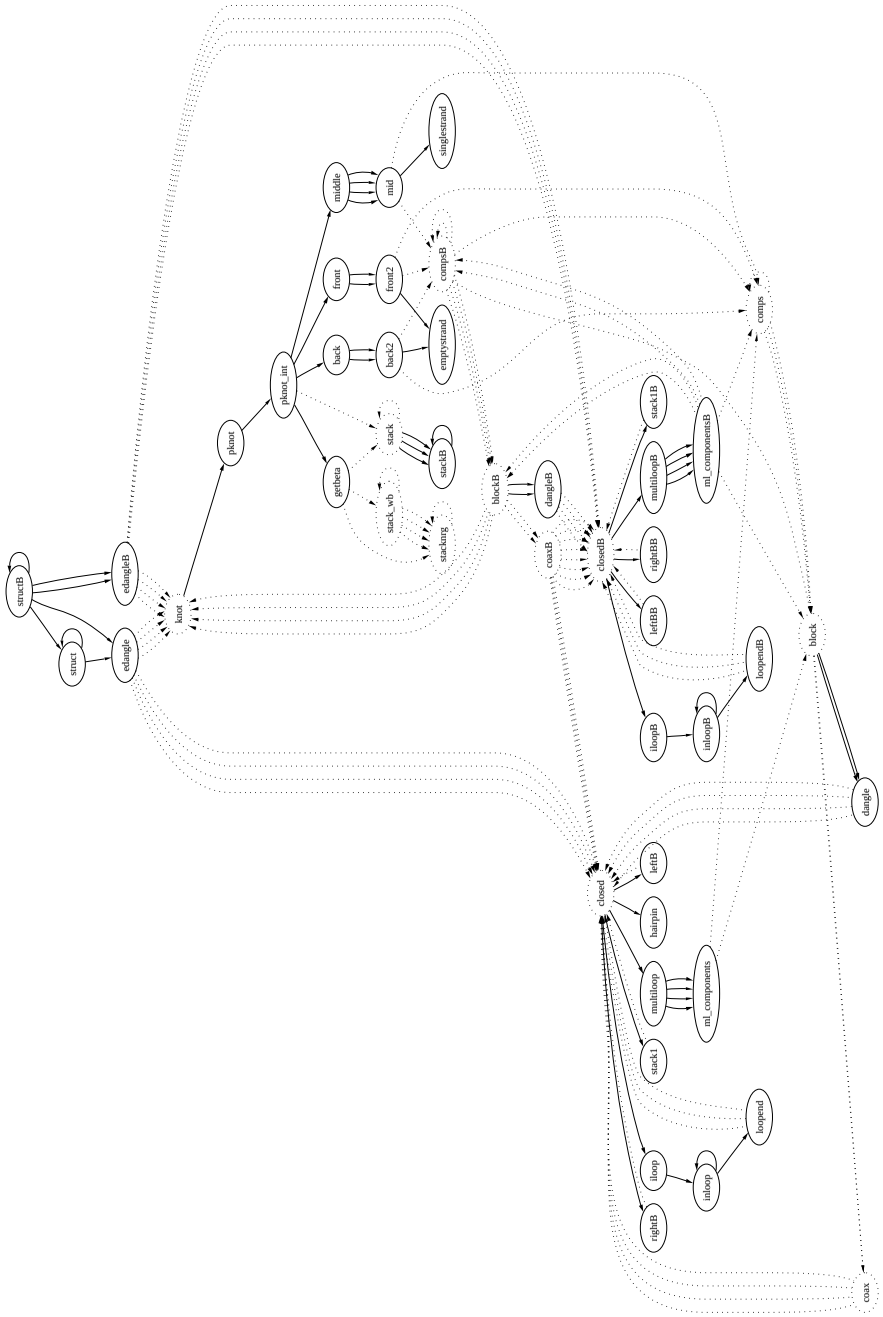


Figure 4.17: Dependence graph for *pknotsRG* – optimal table design (12 tables). This table configuration leads to time and space requirements of $O(n^4)$ and $O(n^2)$, respectively, the same as the manual design, but uses only about 70% of the table space.



```
Usage: pknotsRG [options] <input sequence> (or stdin)
Options:
  -h         Display this information
  -m         Use mfe strategy
  -f         Use enf strategy
  -l         Use loc strategy
  -s         Show suboptimals
  -u         no dangling bases (implies -s)
  -o         no suboptimals inside pknots (implies -s -l)
  -e <value> Set energy range for suboptimals (kcal/mole)
  -c <value> Set energy range for suboptimals (%) [10]
  -n <value> Set npp-value [0.3]
  -p <value> Set pkinit-value [9]
  -k <value> Set maximal pknot-length
  -v         Show version
```

Figure 4.18: pknotsRG interface

- Options used for setting the value of a variable. For example -e for setting the suboptimal energy difference in an RNA program.
- Switches to choose a program mode.
- Switches to choose a boolean value (like -u or -o in the pknotsRG interface).

In the following we describe how to represent such an interface. We used XML as format, since XML is very suitable to describe structured data. Moreover, a huge number of tools and programming frameworks for XML exists, hence it is very convenient to work with XML. Figure 4.19 shows an example.

The top-level element for command line options is the element `option`. This can be deeper specified by the following six elements:

- `help`: gives an overview of the program command line options
- `longhelp`: gives a detailed description of the given command line option
- `version`: shows the current version number
- `switch`: a switch for a boolean variable
- `set`: an option to set the value of a variable
- `switchset`: an option to set a boolean variable and a value at the same time
- `file`: an option for file input

Figure 4.20 shows the C code generated from this XML file. This code uses the `getopt`-library for processing of the command line options.

The XML file can be written by hand or it can be generated by the ADP compiler. Additionally the ADP compiler generates the following

```
<option switch="h" text="Display this information">
  <help />
</option>

<option switch="H" argtext="option" text="Display detailed information on <option>">
  <longhelp />
</option>

<option switch="v" text="Show version">
  <version />
</option>

<option switch="e" argtext="value" text="Set energy range (kcal/mol)">
   <set var="traceback_diff" datatype="float" default="0" minval="0"
</option>

<option switch="w" argtext="value" text="Set window size">
   <switchset var="window_size" swvar="window_mode"
       defaultval="0" default="off" datatype="int" minval="1" />
</option>

<option switch="f" argtext="filename" text="Read input from file">
  <file var="inputfile" />
</option>
```

Figure 4.19: Standard XML file

```
1  // Display detailed information on <option> (-H)
2  case 'H':
3    if      (optarg[0]=='-') { manoptmode = '-'; manopt = optarg[1]; }
4    else if (optarg[0]==':') { manoptmode = ':'; manopt = optarg[1]; }
5    else                     { manoptmode = '-'; manopt = optarg[0]; }
6    if (!interactive) printf("\n");
7    #include "RNAfold2-man.c"
8    if (!interactive) printf("\n");
9    opt->terminate = 1;
10   break;
11 // Show version (-v)
12 case 'v':
13   printf("%s (%s)\n",PACKAGE_STRING,RELEASE_DATE);
14   printf("\n");
15   opt->terminate = 1;
16   break;
17 // Set energy range (kcal/mol) (-e)
18 case 'e':
19   sscanf(optarg,"%f",&(opt->traceback_diff));
20   opt->traceback_diff = max(0, opt->traceback_diff);
21   if (interactive) printf("Energy range set to %.2f kcal/mol.\n", opt->traceback_diff);
22
23   opt->traceback_percent = 0;
24   break;
25 // Set window size (-w)
26 case 'w':
27   if ((interactive) && (optarg[0] == '-')) {
28     printf("Window mode disabled.\n");
29     opt->window_mode = 0;
30   }
31   if (optarg[0] != '-') {
32     opt->window_mode = 1;
33     sscanf(optarg,"%d",&(opt->window_size));
34     opt->window_size = max(1, opt->window_size);
35     if (interactive) printf("Set window size to %d. Type -w - to disable.\n", opt->window_size);
36   }
37   break;
38 // Read input from file (-f)
39 case 'f':
40   if (opt->inputfile) free(opt->inputfile);
41   opt->inputfile = mkstr(optarg);
42   break;
```

Figure 4.20: Example C code for interface

- The C-framework for a complete C-program.
- A file input facility to read text- and fasta-files.
- A sliding window mode for RNA programs.
- An interactive mode. In this mode it is possible to enter input data and command line options in an interactive fashion.
- A Make-file to compile the program. This is especially useful for ADP programmers without knowledge in C programming.

Figure 4.21 shows an overview of the compilation process. Starting point in this example is the file `ElMamun.lhs`. The ADP compiler (`adpc`) generates the XML-file `ElMamun.xml` and the Make-file. The XML-file can then be edited by the user. With the call of make, the adpc is called again, and it generates the main C file `ElMamun.c` and the C-modules `ElMamun_seller.c`, `ElMamun_buyer.c` and `ElMamun_count.c` (one for every algebra). These module files are called from the main file `ElMamun.c`. Finally the Make-file calls the C compiler (`gcc`) to compile the C files and link them into the binary program `ElMamun`.

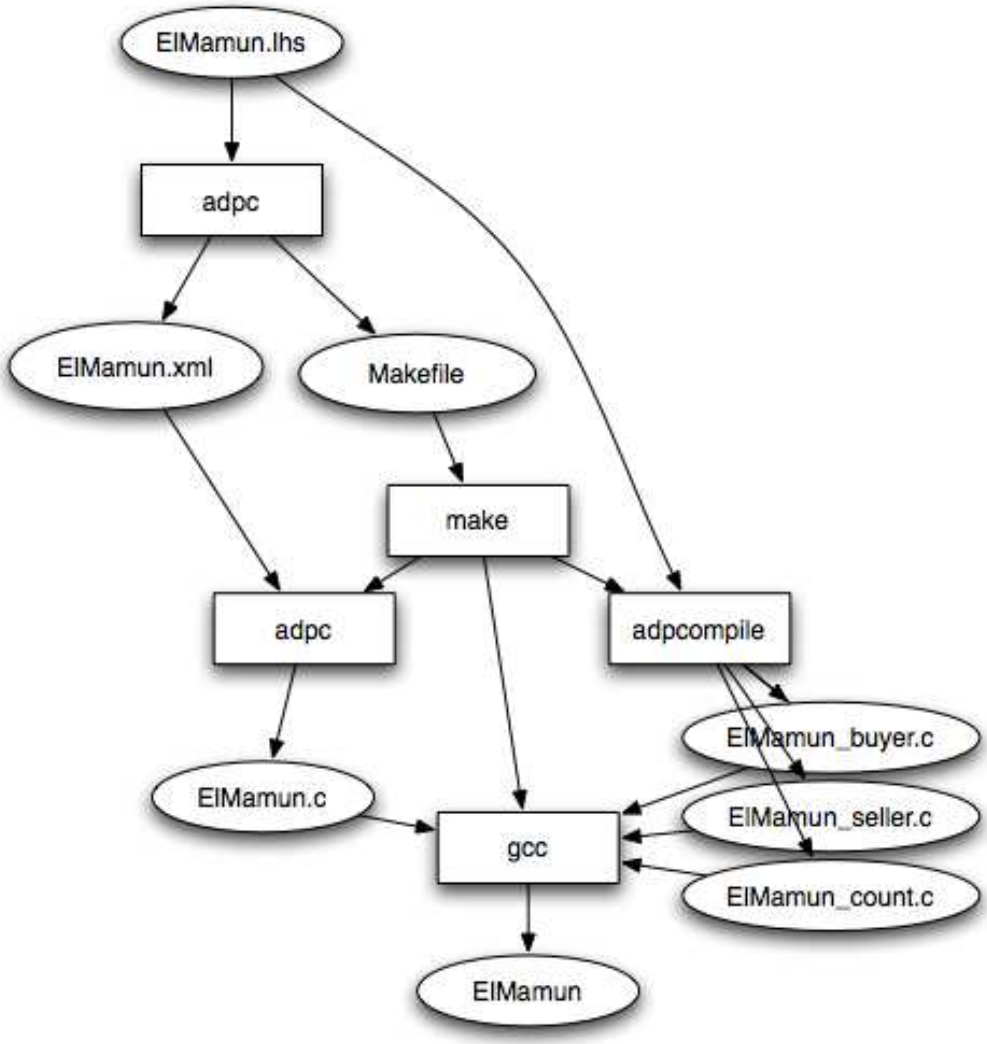


Figure 4.21: Adpc framework

Chapter 5

Applications

This chapter describes RNAshapes, a non-trivial RNA analysis program developed using the ADP method. In Section 5.2 we also give some benchmarks for a number of ADP program compiled with the ADP compiler.

5.1 RNAshapes

This section is taken from [51].

5.1.1 Introduction

Abstraction is our major mental aid to master complexity. When we speak about functional structures of RNA, we speak of long hairpins for miRNA precursors, of clover leaf structures for tRNA, of neighboring hairpins with attenuators, etc., and we often do not care about individual base pairs or helix sizes. For programs comparing RNA structures, it has long been suggested to represent larger structures as trees at different levels of detail [47] (subroutine `b2Shapiro` in the Vienna RNA package [24]).

RNA structure prediction algorithms, however, are ignorant of abstraction and either deceive us with a single, minimum free energy prediction, or overwhelm us with a plethora of near-optimal structures, most of which are very similar and thus redundant.

RNA shape abstraction maps structures to a tree-like domain of shapes, retaining adjacency and nesting of structural features, but disregarding helix lengths. Shape abstraction integrates well with dynamic programming algorithms, and hence it can be applied during structure prediction rather than afterwards. This avoids exponential explosion and can still give us a non-heuristic and complete account of properties of the molecule's folding space.

Rather magically, some long and hard-studied problems become easy.

So far, we have approached three problems with the use of abstract shapes:

1. Computation of a *small* set of representative structures of different shapes, complete in a well-defined sense [20].
2. Computation of accumulated shape probabilities [55].
3. Comparative prediction of consensus structures, as an alternative to the over-expensive Sankoff Algorithm: RNAcast [40].

So far, the first two applications have only been available as implementations in the functional programming language Haskell, with strong limitations on input sequence length. RNAcast was only available as an online tool. Here, we describe a complete reimplementation of these approaches in the programming language C. This new implementation is around 50-100 times faster than the original implementations, has lower memory requirements and can be used with significantly longer input sequences. It combines all three tools in one single package. We also included a number of additional features.

In the following, we will shortly review the notion of abstract shapes, and explain where its power comes from. We will then provide an overview of the problems that can be approached in the new way.

5.1.2 The abstract shapes approach

An RNA shape is an abstract representation of an RNA secondary structure. It is inspired by the dot-bracket representation known from the Vienna RNA package [24]. Consider the following sequence and two secondary structures from its folding space in dot-bracket representation:

```
AUCGGCGCACAGGACAUCCUAGGUACAAGGCCGCCCGUU
..(((.((..(((....))).(((.....)))))))..
..(((.....(((....))).(((.....))).))))..
```

The shapes approach offers five abstraction levels – or *shape types* – ordered in their degree of abstraction. Common to all levels is that they abstract from loop and stack lengths, where unpaired regions are represented by an underscore and stacking regions by a pair of squared brackets. This is the least abstract shape type 1, so the two example secondary structures become:

```
_[_[_[_]_[_]]]_
_[_[_]_[_]_]_
```

The succeeding shape types gradually increase abstraction, ending in type 5, where no unpaired regions are included and nested helices are combined. In this type, our example structures are both represented as:

```
[[][]]
```

These abstractions form the basis of all applications of RNA abstract shape analysis. In the following we give an overview of the main applications, all integrated in the new RNAshapes package.

Shape representative analysis

Current RNA folding algorithms either calculate a single, minimum free energy prediction, or a huge number of suboptimal structures, most of which are quite similar and therefore redundant. With shapes, we abstract from the concrete secondary structures and only consider classes of structures that fall into different shapes. The *shape representative* (in short: *shrep*) of a shape is the structure with the minimum free energy inside a shape class.

Figure 5.1 shows an example program run inside the RNAshapes user interface with the *Natronobacterium pharaonis* tRNA for alanine (gb: AB003409.1/96-167). The predicted *mfe*-structure is one hairpin with internal loops, as depicted in Figure 5.1 on the left. The biologically active structure is the clover-leaf structure (Figure 5.1 right). It would appear at position 123 in the energy sorted list of 308 suboptimals, produced by *RNAsubopt* [57] with an energy range of 5 kcal/mol above the *mfe*. Using RNAshapes, we get three shapes in an energy range of 5 kcal/mol, of which the rank 3 *shrep* is the clover-leaf structure.

Shape probabilities

In [55], we extended the shapes approach to the computation of shape probabilities. The probability of a shape is the sum of the probabilities of all structures that fall into this shape. Several analyses indicate that this approach is quite effective. For example, an analysis of a conformational switch shows the existence of two shapes with probabilities approximately $\frac{2}{3}$ vs. $\frac{1}{3}$, whereas the analysis of a micro RNA precursor reveals the hairpin shape with a probability near to 1.0 [55].

The new implementation contains three approaches for probability analysis, suitable for different input sizes:

Complete probability analysis This implies a complete and non-heuristic analysis of the folding space, where the computational effort depends only on the size of the shape space, which is much smaller than the folding space. On a computer with 2GB main memory, sequences up to a length of around 300 bases can be processed. The following two approaches relax this restriction.

Sampling shapes probability analysis The sampling shapes approach works in the same manner as Ding and Lawrence's Sfold program [6]. In each step of the recursive backtracing procedure, base pairs and the structural element they belong to are sampled according to their probability, which is obtained from the partition function [31]. For each sample, we calculate its corresponding shape. The shape probability then results from its frequency in the sample space. A sample size of 1000 (as also used by the Sfold server) is sufficient for high probability shapes. For details see [55]. The sampling approach is computationally feasible with an input length of up to 1500 bases.

Fast high probability shape analysis The third option only calculates probabilities for shapes with the lowest free energy shreps. These are often also the shapes of highest probability (but not necessarily so). This mode is implemented as a two-step process. In the first step, the lowest free energy shapes are calculated as in *shape representative analysis*. Then, for each of these shapes, the probability is calculated individually. Since these individual calculations have significant lower resource requirements than the complete probability analysis, it is suitable for input sequences up to around 500 bases.

Consensus shapes

The well-known Sankoff algorithm [44] for simultaneous RNA sequence alignment and folding is currently considered an ideal, but computationally over-expensive method. Available tools implement this algorithm under various pragmatic restrictions [30, 22]. See [11] for a recent comparative evaluation of these and several further methods.

In [40], we proposed to redefine the consensus structure prediction problem in a way that does not imply a multiple sequence alignment step. For a family of RNA sequences, our method RNAcast explicitly and independently enumerates the near-optimal abstract shape space, and predicts as the consensus an abstract shape common to all sequences. For each sequence, it delivers the thermodynamically best structure that has this common shape. Since the shape space is much smaller than the structure space, and identification of common shapes can be done in linear time (in the number k of shapes considered), the method is linear in the number s of sequences, yielding $O(n^3 \cdot k \cdot s)$ overall. Our evaluation shows that the new method compares favorably with available alternatives [40]. It is particularly useful on sequences with low conservation, where methods based on sequence alignment cannot be employed. We have now integrated RNAcast into the RNAshapes package.

5.1.3 The RNAshapes package

In addition to the main modes of operation described above, the package offers a number of convenient functions:

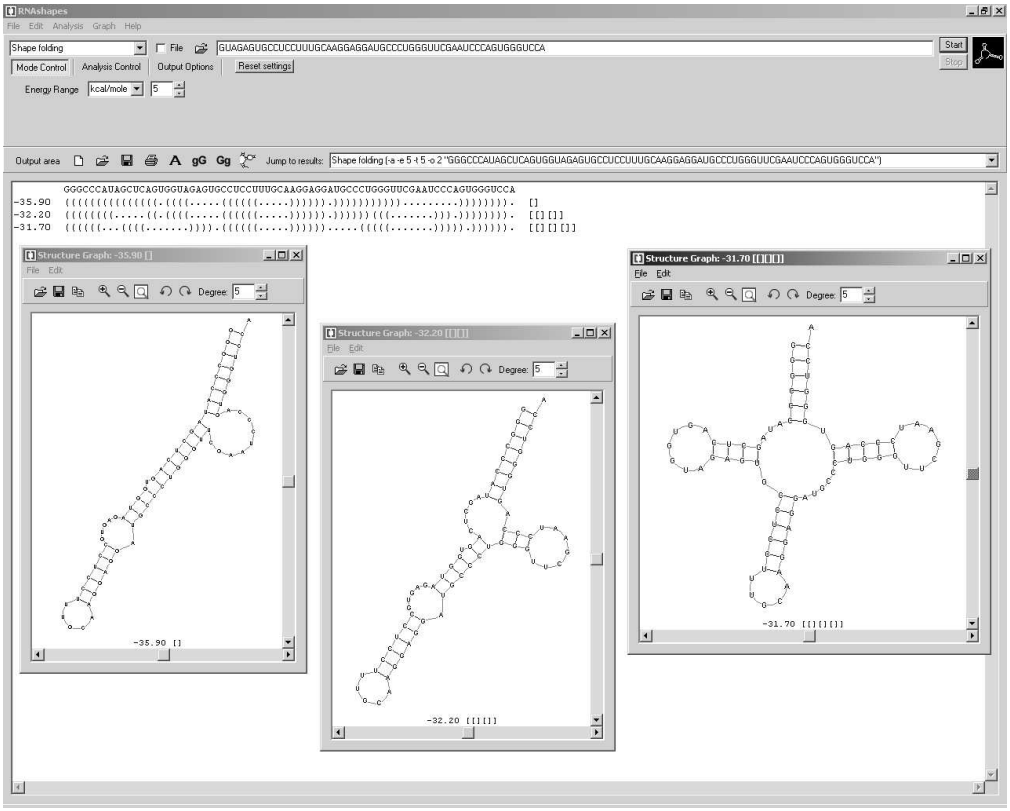


Figure 5.1: Predicted *shreps* for *Natronobacterium pharaonis* tRNA-ala in an energy range of 5 kcal/mol above the *mfe*. This energy range holds 308 structures. The figure shows the RNAshapes user interface for Microsoft Windows. The results of the current RNAshapes analysis are shown in the program's output area. To create a structure graph drawing, the user can simply click onto the dot-bracket string of the desired result.

- Input sequences can either be single sequences, sequence files, or multi-sequence files in fasta format. Additionally, interactive user input of sequences is supported.
- Graphical output of secondary structures in postscript format (implemented with program code from the Vienna RNA package [24]).
- Complete suboptimal folding that handles dangling bases correctly.
- A sliding window function for processing whole genomes. Apart from RNAcast, this option can be used with all analysis modes.
- Detailed options to modify the program output.
- Complete control of program functionality by command line options, useful for automatic script processing.
- A graphical user interface for Microsoft Windows with convenient access to the complete functionality of the package. It also offers an interactive visualization of structures from the program output (Figure 5.1). The appearance of these structure drawings can be controlled in a very flexible manner (e.g. colors, sizes, fonts, orientation).

5.1.4 Conclusion

RNAshapes offers several powerful RNA analysis tools in one single software package. Due to the new implementation in C, it is considerably more efficient than the previous separate implementations. With the integrated graphical user interface the package offers enhanced usability, especially for researchers not used to the command line, and hopefully reaches a new community of users. See Appendix B for an exemplified introduction to the RNAshapes interface.

5.1.5 Implementing the shapes representative analysis

The shape representative analysis is a combination of the three algebras shapes, mfe and pretty. An obvious implementation is the application of the pair operator in the form shape *** (mfe *** pretty). However, this is not suitable, since this combination calculates the shape representatives for *all* shapes. So a hand-written choice function is needed here. One problem of this implementation is that its calculation is very expensive. So for every table element several calculations are necessary. First the shapes must be calculated and then for every shape the corresponding shape representative. For larger inputs this is very time- and space-consuming. Even a calculation based on hash-tables does not really improve efficiency.

Therefore we use a different, three-stage approach for the C implementation.

1. In the first phase, only the results for algebra *mfe* are calculated. Here we have to deal only with atomar results, so the implementation is very efficient (see also Section 4.4.5). After this stage all table entries are filled with their corresponding mfe values.

2. In the second phase, all table entries relevant to the user-given energy range are marked. This is implemented via a variant of the suboptimal backtrace. In this phase all table entries lying on paths of the backtrace are marked and the difference value of the current path is saved. When a cell is already visited, the stored difference value is compared to the current difference value. Only when the current value is larger than the stored one, the suboptimal path is followed further. This strategy is a lot faster than a complete suboptimal backtrace, since most of the suboptimal paths can be interrupted at early stages. After this phase all table entries relevant to the given energy range are marked.

3. In stage 3 now the actual shape analysis is performed. This phase is relatively expensive, but since it only needs to be calculated for the cells marked in phase 2 it is much faster than a complete shape analysis for every cell. The shape analysis is implemented via a hash table where the shape serves as the hash key. So every shape string is stored only once for the complete analysis.

5.2 Benchmarks

In the following, we give some performance results for four bioinformatics tools developed using the ADP method. We used ghc 6.2 (with optimization option `-O2`) to compile the *Haskell* embedded versions, and Sun Studio 10 (with option `-xO5`) for the compiled C versions. The C versions were generated by the ADP compiler. All measurements were performed on an Intel Xeon 2.8 GHz CPU with 2GB main memory, running Solaris 10.

5.2.1 RNAshapes

RNAshapes [20] predicts RNA secondary structures based on the so-called *abstract shapes* method. Its time and space complexity is $O(n^3\alpha^n)$ and $O(n^2\alpha^n)$, where n is the length of the RNA sequence, and α is typically close to 1, but depends on the chosen energy range and shape abstraction.

	ghc		adpc	
n	time (s)	MB	time (s)	MB
200	32.83	58	0.24	14
400	1212.55	684	1.81	22
600	out of memory		6.35	32
800	out of memory		13.96	51

5.2.2 RNAfold

RNAfold calculates secondary structures of RNAs [24]. Its time and space complexity is $O(n^3)$ and $O(n^2)$. We have reimplemented RNAfold in ADP and compared the running times for the three implementations. Column *original* shows the time and space requirements of the original RNAfold implementation.

	ghc		adpc		original	
n	time (s)	MB	time (s)	MB	time (s)	MB
200	18.41	16	0.10	13	0.07	1
400	264.06	90	0.90	14	0.37	3
600	2560.74	601	3.03	16	0.84	4
800	out of memory		7.15	22	1.70	5

5.2.3 pknotsRG

PknotsRG [39] predicts RNA secondary structures including (a subclass of) the so-called pseudoknots. Its time and space complexity is $O(n^4)$ and $O(n^2)$, where n is the length of the RNA primary sequence.

	ghc		adpc	
n	time (s)	MB	time (s)	MB
200	100.46	36	0.93	12
400	1646.19	143	16.17	15
600	8731.38	327	78.16	20
800	28204.53	599	278.74	27

5.2.4 RNAhybrid

RNAhybrid [42] predicts hybridization sites of a short RNA molecule in a long one. Its time and space complexity is $O(s^2 n)$ and $O(sn)$, where n is the length of the long RNA sequence, and s is the length of the short one (kept fixed in the measurements).

	ghc		adpc	
n	time (s)	MB	time (s)	MB
20000	378.88	74	1.79	31
40000	1219.47	143	3.63	57
60000	2556.32	206	5.46	85
80000	4299.91	284	7.19	142

5.2.5 Conclusion

The experiments show that the programs generated by the ADP compiler run at least a hundred times faster than their corresponding ghc versions. For some examples the speedup can even go up to 800 (RNAfold). The memory savings are from factor 2 up to around 35. Compared to the hand-implemented RNAfold program by I. Hofacker et. al. the automatically compiled ADP version is only around 4 times slower and has memory requirements of a factor of around 4.

Chapter 6

Outlook

In this work we have described how to automatically compile ADP programs to programs in an imperative target language like C. On this basis, several other interesting topics can be investigated.

Independence of Haskell. The traditional way to develop an ADP program is the following: First a prototype of the program is developed with the embedded version of the ADP language in Haskell. This prototype is then compiled with the help of the ADP compiler. However, this way has a number of disadvantages (see also Section 1.4). The main disadvantage is that the ADP programmer also has to be a good Haskell programmer. The error messages generated by the Haskell implementations can be quite complex and understanding these messages can also be a challenge for experienced Haskell programmers. With the development of the ADP type checker by Stefanie Schirmer we can now eliminate this problem. Since the type checker knows much more about ADP than a Haskell system, it can generate error messages specific to ADP programs. Another advantage of the independence of Haskell is that we do not depend on the Haskell syntax anymore. We can then define a suitable ADP language on our own. The definition of such an ADP language should be the next step in the development of ADP.

ADP based on multisets. The original formulation of ADP uses *sets of answers* to a DP problem, which are eventually implemented as lists. This creates a certain degree of imprecision when multiplicity of solutions plays a role. Therefore, we will base the ADP theory on *multisets of answers*, which is consistent with an implementation as lists as long as we do not take advantage on their internal order of elements (work with G. Rote, in progress).

Comparing ADP to Algebraic Path Problems. Dynamic programming problems as addressed within the ADP framework can be seen as a special case of the extensively studied algebraic path problems. See [43] for a review, and the recent book by Pachter

and Sturmfels [35], which summarizes applications of the algebraic view to problems in bioinformatics. Algebraic path problems are optimization problems over graphs defined via a semiring structure $(S, \oplus, \odot)$, where S is the domain of scores, $\oplus$ the objective function (such as minimization) applied when joining paths, and $\odot$ is the function that combines scores when extending paths. ADP is somewhat richer semantically, as every function in an evaluation algebra is a separate instance of the $\odot$ operation, which must satisfy the semiring axioms, and may operate on several arcs simultaneously. As a consequence, the optimal "path" is actually a tree. In the graph problem framework, the underlying graph structure is usually given explicitly as input. In the ADP approach, the input is always a sequence, and the graph structure remains implicit. For each problem instance, data dependencies (paths) are determined according to the tree grammar. We are currently investigating which mathematical observations from graph problems carry over to the ADP approach, and which can be seen as extending algebraic graph problems towards easier programming in specialized, but also more sophisticated application domains.

Complete handling of pair-operator. The pair operator is currently only supported for the combination of two algebras. This suffices to compile algebras like `pairmax *** count` or `pairmax *** prettyprint`. The combination of more than two algebras requires some extensions to the compiler. The most common application of the pair operator is the combination of a scoring algebra together with a pretty printing algebra. This is supported by the compiler through the direct generation of backtracing code. This backtracing code is much more efficient than the application of the pair operator. However, a complete support of the pair operator would be very helpful.

ADP in hardware. A field programmable gate array (FPGA) is a semiconductor device containing programmable logic components and programmable interconnects. FPGAs allow to develop own special purpose processors in a cheap and convenient way. FPGAs are used for a large number of applications and also for several bioinformatics projects. An interesting research area now is to what extent the ADP compiler can be extended to automatically generate a description of such an FPGA.

Parallelization. In times of multi-processor computers and multi-core processors, parallelization is an interesting topic. It needs to be examined to what extent ADP programs can automatically be compiled to allow parallel execution.

Code generation for GPUs. A Graphics Processing Unit (GPU) is a dedicated graphics rendering device for personal computers. These GPUs work extremely parallel and can process graphical data much faster than a general purpose CPU. In the last years it has been tried to use this GPUs for other purposes than their original function. The website `http://www.gpgpu.org/` gives an overview of several possible applications. It is an interesting question to what extent these GPUs can be used to execute ADP programs. For this purpose the ADP compiler needs to generate special GPU code.

LaTeX recurrences. Traditionally, DP programs are documented in the form of matrix recurrences. The language of list comprehensions generated during the compilation can be

used to automatically translate the recurrences into LaTeX equations.

Target code backends. The generated target language $\mathcal{TL}$ is very universal. With this language it should be relatively easy to generate other languages than C. Due to its wide spreading Java is certainly interesting. Another possible target language is Fortran. Fortran is widely used in the area of scientific computing and it is said that Fortran programs are quite more efficient than the corresponding C programs.

Appendix A

Proof: $P_{\mathcal{G}}^{\vartheta}$ has polynomial runtime if and only if $\Delta(G) \leq 1$

The number of dependences for a parser p_q is bounded by $|u(q,n)| = O(n^{width(q)})$. Therefore, $r(P_{\mathcal{G}}^{\vartheta}, n)$ can only become exponential in the presence of recursive calls between parser functions (Eq. 4.7).

Definition 9 *(Number of function calls)* $c(q,n)$ *shall denote the overall number of function calls between parser functions initiated by a parser* p_q *on input length* n*. The total number of function calls for a yield parser* $P_{\mathcal{G}}^{\vartheta}$ *on input length* n *is denoted by* $c(P_{\mathcal{G}}^{\vartheta}, n)$*.*

Obviously, the execution time $r(P_{\mathcal{G}}^{\vartheta}, n)$ is polynomial if and only if $c(P_{\mathcal{G}}^{\vartheta}, n)$ is polynomial. In order to distinguish between exponential and polynomial runtimes, we will concentrate on the asymptotic behavior of $c(P_{\mathcal{G}}^{\vartheta}, n)$.

Each edge in the dependence graph represents a function call between two parser functions. A walk π in the graph can therefore be interpreted as a sequence of consecutive function calls where the weight $\tilde{w}(\pi)$ of the walk is the length of the "consumed" input and the length $l(\pi)$ is the number of required function calls. Moreover, the number of walks in a graph can be derived by the powers of the graph's adjacency matrix A. Bringing together these two observations, we get the following result for the asymptotic behavior of $c(P_{\mathcal{G}}^{\vartheta}, n)$.

Lemma 1 *Let* $P_{\mathcal{G}}^{\vartheta}$ *be a tabulating yield parser with* $width(\mathcal{G}) = 0$ *and corresponding dependence graph* G *and adjacency matrix* A*. The asymptotic number of parser function calls initiated by* $P_{\mathcal{G}}^{\vartheta}$ *on input length* n *is given by*

$$c(P_{\mathcal{G}}^{\vartheta}, n) = O(n \cdot N(A^n)) \quad \text{with} \quad N(A) = (\sum_{i,j=1}^{s} |a_{ij}|^2)^{1/2} \ . \tag{A.1}$$

Proof. Wlog. we assume that each parser terminates at input length 0. Since $\tilde{w}(\pi)$ may not exactly sum up to an input length n, we define the following approximation relation: For $\pi = v_0 e_1 v_1 e_2 \dots e_l v_l$, $\tilde{w}(\pi) \approx n$ iff $\tilde{w}(\pi) \geq n$ and $\tilde{w}(v_0 e_1 v_1 e_2 \dots e_{l-1} v_{l-1}) < n$.

Let n be the input length and π a walk of weight $\tilde{w}(\pi) \approx n$ starting at vertex q. Then, π represents one of possibly many chains of function calls started by parser p_q on input length n with the number of calls given by $l(\pi)$. Defining W_q as the set of walks starting in q, we get $c(q,n) = \sum_{\pi \in W_q} \{l(\pi) | \tilde{w}(\pi) \approx n\}$. Since all $w(e_i)$ are constant and independent of n, for a walk π with $\tilde{w}(\pi) \approx n$ also holds $l(\pi) = O(n)$. Therefore, we can equally state: $c(q,n) = O(\sum_{\pi \in W_q} \{l(\pi) | l(\pi) = n\})$. This eliminates the need for edge weights in the dependence graph.

It is well known that the n-th power of A can be used to determine the number of distinct walks in G, such that $a_{ij}^{(n)}$ holds the number of walks from vertex i to vertex j of length n [5, Th.1.9]. This implies $c(q,n) = O(n \sum_{j=1}^{s} a_{qj}^{(n)})$. We assume that all nonterminal symbols of $P_{\mathcal{G}}^{\vartheta}$ are reachable from the axiom. Therefore, each element $a_{ij}^{(n)}$ represents a number of function calls affecting the runtime of $P_{\mathcal{G}}^{\vartheta}$. Then the following holds: $n \cdot \max_{i,j} a_{ij}^{(n)} \leq c(P_{\mathcal{G}}^{\vartheta}, n) \leq n \sum_{i,j=1}^{s} a_{ij}^{(n)}$. Since $\max_{i,j} a_{ij}^{(n)} = O(\sum_{i,j=1}^{s} a_{ij}^{(n)})$, it follows that $c(P_{\mathcal{G}}^{\vartheta}, n) = O(n \sum_{i,j=1}^{s} a_{ij}^{(n)})$.

$N(A)$ denotes the Frobenius norm of the matrix A which is defined by $N(A) = (\sum_{i,j=1}^{s} |a_{ij}|^2)^{1/2}$. It follows from $\sum_{i,j=1}^{s} |a_{ij}| = O(N(A))$ that $c(P_{\mathcal{G}}^{\vartheta}, n) = O(n \cdot N(A^n))$. □

With Lemma 1 we can now analyze the runtime of $P_{\mathcal{G}}^{\vartheta}$ by examining the asymptotic behavior of the powers of the corresponding dependence graph's adjacency matrix. A suitable result was given by W. Gautschi in 1953. For convenience, we state his main theorem:

Theorem 6 (W. Gautschi [12]) *Let A be a (real or complex) $s \times s$ matrix and suppose that not all eigenvalues λ_v of A are zero. Denote by m_v the multiplicity of λ_v in the minimal polynomial of A and put*

$$k = \max_{v=1,\dots,s; \lambda_v \neq 0} m_v$$

Then we have for a constant $c > 0$ depending only on A:

$$1 \leq \frac{N(A^n)}{(\sum_{v=1}^{s} |\lambda_v|^{2n})^{\frac{1}{2}}} \leq c n^{k-1} \qquad (n = 1, 2, \dots) \tag{A.2}$$

If all λ_v are zero, we have $N(A^n) = 0$ $(n \geq l, l = m_v)$.

Here $|\cdot|$ denotes the complex modulus defined by $|x + iy| = \sqrt{x^2 + y^2}$. Following this result, the eigenvalues λ_v have the largest impact on the asymptotic behavior of $N(A^n)$. For our purposes we will distinguish between three cases. The maximal multiplicity m_v of

an eigenvalue λ_v of A is s, thus:

$$N(A^n) = \begin{cases} 0 & \text{if all } \lambda_v = 0 \\ O(n^{s-1}) & \text{if all } |\lambda_v| \leq 1 \\ O(x^n) & \text{if any } |\lambda_v| > 1 \ (x > 1) \end{cases} \tag{A.3}$$

Given that $c(P_{\mathcal{G}}^{\vartheta}, n) = O(n \cdot N(A^n))$, the existence of an eigenvalue $|\lambda_v| > 1$ of the dependence graph's adjacency matrix makes the difference between a polynomial and an exponential runtime. To characterize graphs whose adjacency matrices have eigenvalues of the properties stated in (A.3) we follow results given in [5] and [27]. We summarize them with a short proof:

Lemma 2 *Let G be a directed graph with multiple edges and loops and let λ_1 be the eigenvalue of the adjacency matrix A of G with the largest modulus. Then,*

1. $\Delta(G) = 0$ iff $\lambda_1 = 0$.
2. $\Delta(G) = 1$ iff $\lambda_1 \leq 1$.
3. $\Delta(G) > 1$ iff $\lambda_1 > 1$.

Proof. Since A is non-negative, λ_1 is real and non-negative [54, Th. 2.7]. The trace $Tr(A)$ of a matrix A is the sum of its diagonal elements. It is well known that $Tr(A) = \sum_{i=1}^{s} \lambda_i$ and $Tr(A^n) = \sum_{i=1}^{s} \lambda_i^n$ (see e.g. [10, IV §5]). If G contains no circuit at all, then $Tr(A^n) = 0$ for any n. This proofs statement 1. If $\Delta(G) = 1$, then $Tr(A^n) \leq s$ for any n. Therefore $\sum_{i=1}^{s} \lambda_i^n \leq s$ for any n which proofs statement 2. If $\Delta(G) > 1$, then $Tr(A^n)$ *must* increase with n. Therefore, an eigenvalue $\lambda_i > 1$ must exist, since otherwise $\sum_{i=1}^{s} \lambda_i^n \leq s$. □

Figure A.1 shows one example graph for each of the three cases. Altogether, we get the following result, connecting the runtime of a tabulating yield parser with the cyclic structure of its dependence graph:

Theorem 7 *Let $P_{\mathcal{G}}^{\vartheta}$ be a tabulating yield parser with corresponding dependence graph G. Then, $P_{\mathcal{G}}^{\vartheta}$ has polynomial runtime if and only if $\Delta(G) \leq 1$.*

Proof. It follows from Lemmas 1,2 and Theorem 6 that $c(P_{\mathcal{G}}^{\vartheta}, n)$ is polynomial if and only if $\Delta(G) \leq 1$. Theorem 2 implies that the runtime of $P_{\mathcal{G}}^{\vartheta}$ is polynomial if and only if $c(P_{\mathcal{G}}^{\vartheta}, n)$ is polynomial. □

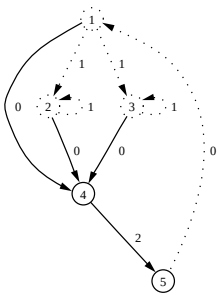
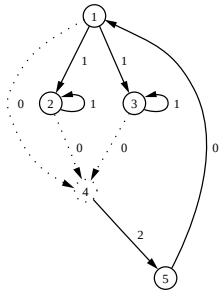
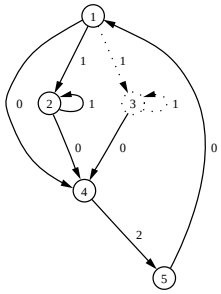
G			
ϑ	$\{1,2,3\}$	$\{4\}$	$\{3\}$
$\Delta(G)$	0	1	2
A	$\begin{vmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{vmatrix}$	$\begin{vmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{vmatrix}$	$\begin{vmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{vmatrix}$
λ_1	0	1	1.47
$N(A^{100})$	0	2.83	$4.55 \cdot 10^{16}$
$r(P^{\vartheta}_{\mathcal{G}}, n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$

Figure A.1: Three dependence graphs for the yield grammar of Example 4.5.1, with corresponding table configurations ϑ, circuit degrees $\Delta(G)$, adjacency matrices A, largest eigenvalues λ_1 and the matrix norms $N(A^{100})$. The vertices are numbered as follows: $1 \rightarrow$ `pal`, $2 \rightarrow$ `del`, $3 \rightarrow$ `ins`, $4 \rightarrow$ `match`, $5 \rightarrow$ `inner`. Table access dependences are drawn as dotted edges. Note that the edge weights are not represented in the adjacency matrices. The runtime of the yield parser $P_{\mathcal{G}}$ under table configuration ϑ is denoted by $r(P^{\vartheta}_{\mathcal{G}}, n)$.

Appendix B

RNAshapes

B.1 The RNAshapes interface by example

In the following, we give an exemplified introduction to the RNAshapes interface. All sequence files used here can be found in the directory `examples` of the RNAshapes distribution[1].

B.1.1 Shape representative analysis

As an example sequence, we use the alanine tRNA of Natronobacterium pharaonis (gb: AB003409.1/96-167). It is contained in sequence file `pharaonis.seq` in the examples directory. With option **-f**, we specify the name of the input file for RNAshapes. When given no additional parameters, RNAshapes works in shape representative analysis mode:

```
RNAshapes -f pharaonis.seq
```

Alternatively, we can also call RNAshapes directly with the sequence:

```
RNAshapes GGGCCCAUAGCUCAGUGGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUGGGUUCGAAUCCCAGUGGGUCCA
```

This calls both give the following result:

```
       GGGCCCAUAGCUCAGUGGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUGGGUUCGAAUCCCAGUGGGUCCA
-35.90 ((((((((((((((((.((((.....((((((.....))))))..))))))))))).........)))))))).  []
```

[1] Download from `http://bibiserv.techfak.uni-bielefeld.de/rnashapes/`.

The predicted mfe-structure is one hairpin with internal loops. In default setting, RNAshapes considers the energy range 10% above the mfe. In this example, all structures in this energy range (-35.90 to -32.31) fall into the hairpin shape `[]`, and the printed structure is the best (i.e. with lowest energy) structure of this shape class. This is the shape representative structure (in short: *shrep*).

To increase the energy range, we can use the switches **-e** (for absolute difference above the mfe), or **-c** (for a percent value above the mfe). Let's try with option **-e** 5:

```
RNAshapes -f pharaonis.seq -e 5
        GGGCCCAUAGCUCAGUGGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUGGGUUCGAAUCCCAGUGGGUCCA
-35.90  ((((((((((((((.((((.....((((((.....)))))).)))))))))).........)))))))).  []
-32.20  ((((((((.....((.((((.....((((((.....)))))).))))))(((.......))).)))))))).  [[][]]
-31.70  ((((((...((((.......)))).((((((.....))))))).....(((((.......))))).))))))).  [[][][]]
```

This gives three results. The biologically active structure of the alanine tRNA is the cloverleaf structure, the third result of the above call.

By default, RNAshapes uses the most abstract shape type (type 5). The shape type can be changed with option **-t**. For example, the same analysis with shape type 3 gives the following results:

```
RNAshapes -f pharaonis.seq -e 5 -t 3
        GGGCCCAUAGCUCAGUGGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUGGGUUCGAAUCCCAGUGGGUCCA
-35.90  ((((((((((((((.((((.....((((((.....)))))).)))))))))).........)))))))).  [[[[]]]]
-33.20  ((((((((((((((.(((...((.((((((.....))))))))))))))))).........)))))))).  [[[[[]]]]]
-32.20  ((((((((.....((.((((.....((((((.....)))))).))))))(((.......))).)))))))).  [[[[]]][]]
-31.90  ((((((..........((((.....((((((.....)))))).))))(((((.......))))).))))))).  [[[]][]]
-31.70  ((((((...((((.......)))).((((((.....))))))).....(((((.......))))).))))))).  [[][][]]
```

With option **-Z**, we can tell RNAshapes to color all dotbracket and shape strings in the output such that corresponding structural elements have the same color in both representations. This is especially useful for examining and learning the different shape types.

```
RNAshapes -f pharaonis.seq -e 5 -t 3 -Z
```

To get a more visual impression, we can also generate structure graphs of these results with option **-g**. As parameter, **-g** receives the number of graphs that should be generated:

```
RNAshapes -f pharaonis.seq -e 5 -g 3
```

This call generates the postscript files `rna_1.ps`, `rna_2.ps`, and `rna_3.ps` (Figure B.1).

So far, we have only entered a single sequence as input for RNAshapes. In practice, multiple sequence files are often more convenient. RNAshapes supports the fasta file format. We call RNAshapes with option **-f** and the example file `random.fasta`, that contains two random sequences:

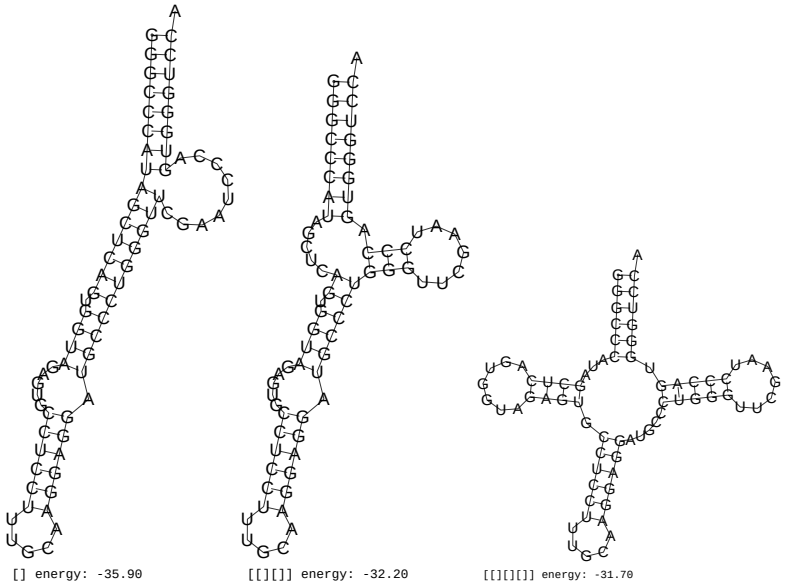


Figure B.1: Postscript structure graphs generated with option **-g**

```
RNAshapes -f random.fasta
>random1
        AGUCUUGACGGUCGCAGCCGUCUCGCGUGGGGCCGCGUCUUUAGGGAUCCAAGGCAUGAGAAUUGAUCGU
-19.30  .((((((.(((((.((((......)).)).)))))).((((....)))).))))))..............  [[][]]
-18.70  .......(((((((((((((((((((....)))))).....)))))....))).))).....)))))))  []
-18.60  .((((((.(((((.((((......)).)).)))))).((((....)))).))))))..((.......)).. [[][]][]
>random2
        UUUGAAUAUCAGCUCAACAGGCAACGGUCCGGUGUAAUCUUUCUGUUAUGGCAGUUAUACCCCGGGAUGAUCACCGCAACUGCGUAGGGU
-20.60  ..(((.((((.(((.....)))..(((...(((((((.....((((....))))))))))))))).)))).))).(((....))).....  [[][]][]
-19.00  ...........((((.........((..((((.(((......((((....))))...))).)))))..)).....(((....))).))))  [[][]]
```

The sequences in the input file are processed one after another, and the results are printed together with the corresponding sequence descriptor.

We can also feed RNAshapes by standard input, which is useful for directly processing sequence output generated by other programs. For example, the call

```
cat random.fasta | RNAshapes
```

is equivalent to the **-f** call above. Finally, we can also enter sequences interactively by calling RNAshapes without any input sequence (but possibly with parameters):

```
RNAshapes -e 5
```

```
Interactive mode. Try 'RNAshapes -h' for more information.

Input sequence (upper or lower case); :q to quit, -h for help.
....,....1....,....2....,....3....,....4....,....5....,....6....,....7....,....8
```

The interactive mode supports the following features:

- Direct input of sequences (or alternatively by cut-and-paste).
- Input history with keyboard keys `UP` and `DOWN` (library `editline` required).
- Complete interface to all program options. Instead of a sequence, simply type one or several RNAshapes commands. Examples:
  - `-h` shows the command overview.
  - `-H <option>` shows a detailed help for the given option.
  - `-c 20` sets the energy range to 20%.
  - `-C -f ires.fasta` switches to consensus shapes mode and starts the analysis with file `ires.fasta`.
  - `:s` shows the current settings.

B.1.2 Shape probabilities

RNAshapes offers a number of options to calculate structure and shape probabilities. The probability of a shape is the sum of the probabilities of all structures that fall into this shape.

The probability analysis modes are controlled with several options (**-p**, **-q**, **-P**, **-i**, **-r**, **-F**, **-T**). The most convenient (but computational most expensive one) is **-q**, the complete probability analysis with simultaneous shrep calculation. Applying this to the alanine tRNA from above gives the following result:

```
RNAshapes -q -f pharaonis.seq
        GGGCCCAUAGCUCAGUGGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUGGGUUCGAAUCCCAGUGGGUCCA
-35.90  ((((((((((((((.((((.....((((((.....))))))..)))))))))))).........)))))))).  0.9897545  []
-32.20  ((((((((.....((.((((.....((((((.....))))))..))))))(((.......))).)))))))).  0.0089891  [[][]]
-31.70  ((((((...((((.......)))).((((((.....))))))).....(((((.......)))))..))))))).  0.0012534  [[][][]]
-27.70  ((((((.((((((.......)))).((((((.....))))))..))..(((((.......)))))..))))))).  0.0000029  [[[][]][]]
```

The shape holding the mfe-structure, `[]`, is also the most probable one (98.97545%), whereas the other shapes have probabilities below 1%. This means, that this unmodified tRNA is very unlikely to occur in the clover-leaf shape (this is consistent with biological knowledge and clearly expresses the need for other mechanisms, such as base modifications, to ensure that the clover-leaf structure is actually achieved).

As a second example, we use the pheS-pheT-Attenuator of E. coli (embl:V00291.1/3682-3754). It is known to switch from a translationally inactive to a translationally active conformation under specific conditions. These two conformations correspond to two valleys in the structure landscape that are separated by a saddle point (energy barrier). In terms of shape analysis, this means that two shapes with reasonable probability should be present. The corresponding experiment delivers the following results (example file `attenuator.fasta`):

```
RNAshapes -q -f attenuator.fasta
> pheS-pheT-Attenuator of E. coli (embl:V00291.1/3682-3754)
        ATCCAGGAGGCTAGCGCGTGAGAAGAGAAACGGAAAACAGCGCCTGAAAGCCTCCCAGTGGAGGCTTTTTTTG
-21.20  ...((((..(((....(((..........))).....))).)))).((((((((....))))))))......  0.5381946  [][]
-20.93  .((((((((((.((((((.................)).))))).....)))))))...))))...........  0.3243870  []
-19.33  ...(((((.....((((((.................)).))))).....((((((((((....))))))))))))  0.0975740  [][]]
-19.73  ..((....))...((((((.................)).))))..((((((((((....))))))))))....  0.0388666  [][][]
-17.30  ...((((..((....)).............((........)))))).((((((((....))))))))......  0.0008492  [][]][]
-15.13  ..((....)).(((((((.................)).))))..((((((((((....))))))))..))  0.0001033  [][][]]
-14.20  ...(((((.((....)).............((.........))..((((((((....))))))))))))  0.0000243  [][][]]
-13.10  ..((....))......(((..........)))).....(((...))).((((((((....))))))))......  0.0000010  [][][][]
```

The shape with two hairpins has a probability of 0.5381946, whereas the one hairpin shape has a probability of 0.3243870. Shape 1 corresponds to the "off" position of the switch and the higher probability resembles that this is its native position. The "on" position (Shape 2) is less probable, expressing the need for some external effect to trigger the switch.

As stated above, RNAshapes offers several options to control probability calculation, where option **-q** is the computational most expensive one. This is due to the fact, that this analysis performs an analysis of the complete folding space. Although the computational effort depends only on the size of the shape space (which is much smaller than the complete folding space), it requires a substantial amount of computer main memory. In our experience, option **-q** can be used with sequences up to 250 bases on a computer with 2GB main memory. Additionally, this restriction depends not only on the sequence length, but also on the structural properties of the sequence. To get an impression of the expected running time for a certain calculation, we can use the option **-B** that shows a progress bar:

```
RNAshapes -q -B -f attenuator.fasta
```

A slightly more efficient variant of option **-q** above is option **-p**. This performs the same analysis, but without calculation of the corresponding shreps. This option should work with sequences up to a length of around 300 bases (for the presentation, we only use the "short" attenuator sequence here):

```
RNAshapes -p -f attenuator.fasta
> pheS-pheT-Attenuator of E. coli (embl:V00291.1/3682-3754)
ATCCAGGAGGCTAGCGCGTGAGAAGAGAAACGGAAAACAGCGCCTGAAAGCCTCCCAGTGGAGGCTTTTTTTG
0.5381946  [][]
0.3243870  []
```

```
0.0975740  [[][]]
0.0388666  [][][]
0.0008492  [[][]][]
0.0001033  [][[][]]
0.0000243  [[][][]]
0.0000010  [][][][]
```

Now we take a look at option **-P**. This option only calculates probabilities for shapes with the lowest free energy shreps. These are often also the shapes of highest probability (but not necessarily so). This mode is implemented as a two-step process. In the first step, the lowest free energy shapes are calculated as in shape representative analysis. Then, for each of these shapes, the probability is calculated individually. The parameter for **-P** specifies the number of lowest free energy shapes for which the probability should be calculated:

```
RNAshapes -P 5 -f attenuator.fasta
> pheS-pheT-Attenuator of E. coli (embl:V00291.1/3682-3754)
        ATCCAGGAGGCTAGCGCGTGAGAAGAGAAACGGAAAACAGCGCCTGAAAGCCTCCCAGTGGAGGCTTTTTTTG
-21.20  ...((((..(((....(((..........)))......))).)))).(((((((((....)))))))).....  0.5381930  [][]
-20.93  .((((((((((.(((((..................)).)))))....)))))))...))))..........  0.3243860  []
-19.33  ...(((((.....(((((..................)).)))))...(((((((((....)))))))))))))  0.0975737  [[][]]
-19.73  ..((....))...(((((..................)).)))))..((((((((((....))))))))))....  0.0388665  [][][]
(Only 4 of 5 probabilities calculated. To get more results, increase energy range (-e or -c).)
```

By default, the lowest free energy shapes are calculated for an energy range up to 10% above the mfe. In cases, where this energy range contains fewer shapes than requested with the **-P** option, we have to increase the energy range with options **-e** or **-c**. In our experience, option **-P** works with sequences up to 500 bases.

The last, and most powerful approach for long sequences, is the sampling shapes probability analysis. The sampling shapes approach works in the same manner as Ding and Lawrence's Sfold program. In each step of the recursive backtracing procedure, base pairs and the structural element they belong to are sampled according to their probability, which is obtained from the partition function. For each sample, we calculate its corresponding shape. The shape probability then results from its frequency in the sample space. The sampling shapes approach is activated with option **-i**, together with the desired number of samples:

```
RNAshapes -i 1000 -f attenuator.fasta
... samples ...
Results for 1000 iterations:

-21.20  ...((((..(((....(((..........)))......))).)))).(((((((((....)))))))).....  0.5050000  [][]
-20.93  .((((((((((.(((((..................)).)))))....)))))))...))))..........  0.3560000  []
-19.33  ...(((((.....(((((..................)).)))))...(((((((((....)))))))))))))  0.0990000  [[][]]
-19.73  ..((....))...(((((..................)).)))))..((((((((((....))))))))))....  0.0390000  [][][]
-13.60  .........((....)).....((((((..(((..........))))..(((((((....)))))))))))).  0.0010000  [][[][]]
```

Our experiments revealed that 1000 samples are sufficient to achieve reasonable results (confidence value of 95% allowing 10% deviation) for shapes with high probability and that

this number is independent of the sequence length (details in [55]). The sampling approach is feasible with an input length of up to 1500 bases.

Finally, we can also calculate the probabilities of individual structures by adding the option **-r** to the program call. This option can be used with any analysis mode of RNAshapes. For example, with our first analysis from above, we receive the following result:

```
RNAshapes -f pharaonis.seq -e 5 -r
                    GGGCCCAUAGCUCAGUGGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUGGGUUCGAAUCCCAGUGGGUCCA
-35.90  (0.4012148)  ((((((((((((((.((((.....((((((.....)))))).)))))))))))).........)))))))).  []
-32.20  (0.0009912)  (((((((.....((.((((.....((((((.....)))))).)))))))(((.......))).)))))))).  [[][]]
-31.70  (0.0004404)  ((((((...((((.......)))).((((((.....))))))).....((((.......)))))..))))))).  [[][][]]
```

The structure probabilities are printed in the second column. Here, the mfe structure has a probability of 40.12%.

B.1.3 Consensus shapes analysis

RNAshapes integrates the RNAcast approach for prediction of consensus structures [40]. For a family of RNA sequences, the method independently enumerates the near-optimal shape space, and predicts as the consensus an abstract shape common to all sequences. As an example, we use a family of IRES elements of Picornaviridae viruses (sequence file `ires.fasta`). The consensus shape mode is activated with option **-C**:

```
RNAshapes -C -f ires.fasta
1) Shape: [][][]]  Score: -223.50  Ratio of MFE: 0.99
> EMCBCG
        CCUUUGCAGGCAGCGGAAAUCCCCACCUGGUAACAGGUGCCUCUGCGGCCAAAAGCCACGUGUAUAAGAUACACCUGCAAAGG
-34.10  ((((((((((..(((((......((((((....))))))..))))))(((.....)))..((((.....)))).)))))))))  [][][]]  R = 2
> MNGPOLY
        CCUUUGCAGGCAGCGGAAUCCCCCACCUGGUGACAGGUGCCUCUGCGGCCGAAAGCCACGUGUGUAAGACACACCUGCAAAGG
-39.10  ((((((((((..(((((......((((((....))))))..))))))(((.....)))..((((.....)))).)))))))))  [][][]]  R = 2
> FDI251473
        GCACGCAAGCCGCGGGAACUCCCCCUUGGUAACAAGGACCCGCGGGGCCGAAAGCCACGUUCUCUGAACCUUGCGUGU
-34.10  ((((((((((((((((.......(((((....)))))).)))))))))(((.....)))..((((...))))))))))  [][][]]  R = 2
> FMDVALF
        GCAUGAUGGCUGUGGGAACUCCCCCUUGGUAACAAGGACCCACGGGGCCAAAAGCCACGUCCUCACGGACCCAUCAUGC
-34.70  ((((((((((((((((.......(((((....)))))).)))))))))(((.....)))..((((....))))))))))  [][][]]  R = 2
> FAN133359
        GCAUGACGGCCGUGGGAACUCCUCCUUGGUAACAAGGACCCACGGGGCCAAAAGCCACGCCCACACGGGCCCGUCAUGU
-41.90  ((((((((((((((((.....((((((....))))))))))))))))(((.....)))..((((....))))))))))  [][][]]  R = 1
> PIFMDV2
        GCAUGUUGGCCGUGGGAACACCUCCUUGGUAACAAGGACCCACGGGGCCGAAAGCCAUGUCCUAACGGACCCAACAUGU
-39.60  ((((((((((((((((.....((((((....))))))))))))))))(((.....)))..((((....))))))))))  [][][]]  R = 1
```

The consensus shapes are printed in order of their score. Here, the score is the sum of the shrep energies. The "ratio of MFE"-value specifies the ratio of this score to the sum of the

minimum free energies. A ratio near 1.0 means a good conservation, a lower ratio means less conservation. The R-value is the rank of the shape in the shape space of the corresponding sequence.

As in all other analysis modes, the shape type can be controlled with option **-t**. For consensus shape analysis, we generally prefer to work with the less abstract level 3:

```
RNAshapes -C -f ires.fasta -t 3
```

To get more results, we can also increase the energy range with options **-e** and **-c**, for example:

```
RNAshapes -C -f ires.fasta -t 3 -e 10
```

We propose to use the output of the consensus shapes analysis as input for RNAforester [23], a multiple RNA structure alignment program. Use output type **-o** f together with option **-C** to generate suitable input for RNAforester. For example:

```
RNAshapes -C -f ires.fasta -o f | RNAforester -m
```

Note that with output type **-o** f only the result for the first consensus is printed (otherwise RNAforester would not work properly). Use the shape match option **-m** to get alternative results. RNAforester is now part of the Vienna RNA package and can be downloaded at `http://www.tbi.univie.ac.at/~ivo/RNA/`.

B.1.4 Additional options

Sliding window mode. Apart from consensus shapes, each analysis mode can be used with a sliding window mechanism. Here, the complete input sequence is processed by individual calculations of subsequences of the specified window size.

For example, the file `V00291` contains the E.coli thrS, infC, rplT, pheS, pheT and himA genes (embl:V00291.1) and has a length of 7784 bases. To start the probability analysis with a window size of 73, we call (this will take some time, maybe stop the calculation with `ctrl-c`):

```
RNAshapes -q -f V00291 -w 73
>V00291 V00291.1 12-SEP-2005
        1                                                                  73
        gattcagtttatgctgctgtaaatccgctcgcagtaaacctttcagacgcaccgtgatgttatcagttgttct
-8.80   ...........((((((.((......)).))))))........(((.(((...((((...)))).))))))  0.6994365  [][]
-8.20   (((.((.....((((((.((......)).))))))...........(((...))).)).)))........  0.2448787  [[][]]
-7.50   ((....(((..((((((.((......)).))))))))))...))..........................  0.0425042  []
-5.70   ((......)).((((((.((......)).))))))........(((.(((...((((...)))).))))))  0.0101577  [][][]
-4.90   ((..(((....((((((.((......)).))))))...........(((...))).((....)).))).)).  0.0014580  [[][][]]
```

```
-4.40   ...........((((((.((......)).))))))........((((((...))).((....)).....)))  0.0007761  [][[][]]
-4.30   ......((...((((((.((......)).))))))...((...)).))....((((...))))........  0.0005654  [[][]][]
-3.30   ((.........((((((.((......)).))))))..(((..(((...))).((....)).)).)).  0.0001179  [[][[][]]]
-3.60   ((....((...((((((.((......)).))))))...((...)).))....((((...)))).....)).  0.0000690  [[[][]][]]
-2.40   ((......)).((((((.((......)).))))))..........(((...))).((....)).....  0.0000288  [][][][]
-1.30   ((......)).((((((.((......)).))))))........((((((...))).((....)).....)))  0.0000046  [][][[][]]
-1.00   ((..((((....))))(((((.........)))).....)).((((...))).((....)).....  0.0000021  [[][]][][]
-1.30   ((..(((....((((((.((......)).))))))...((...)).(((...))).((....)).))).)).  0.0000010  [[][][][]]
```

This is the result for the first "window", the bases 1-73. After this, the window is moved by one base, and the calculation continues with bases 2-74, and so on. Note that these succeeding calculations are faster than the first one, since only a single column of the dynamic programming matrices has to be calculated in each window step. The window step size can be changed with option **-W**.

The result for each window is the same as if we would calculate the corresponding subsequence individually. For example, the window 3682-3754 gives the same result as our pheS-pheT-Attenuator analysis from above, as this is exactly the same sequence.

It would be nice to extend the window approach in a way that the results for the individual windows are merged to a result for the complete sequence. Currently, this is not implemented in RNAshapes.

Suboptimal folding. RNAshapes offers a complete suboptimal folding mode. Its implementation is based on a nonambiguous RNA grammar and handles dangling energies correctly. It is activated with option **-s**:

```
RNAshapes -f pharaonis.seq -s -e 5
        GGGCCCAUAGCUCAGUGGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUGGGUUCGAAUCCCAGUGGGUCCA
-31.10  (((((((((....(((.((((.....(((((((...)))))))..)))))))((........)).)))))))).  [][]
-31.10  (((((((((....(((.((((.....(((((.....)))))).)))))))((........)).)))))))).  [][]
-32.20  (((((((((.....((.((((.....(((((((...)))))))..)))))))(((.......))).)))))))).  [][]
.... (69 results more) ...
```

Shape matching. To see only those structures, that fall into a certain shape, we can use the "shape match" option **-m**. For example, to see all clover-leaf structures in the energy range 5 kcal/mol above the mfe, we call:

```
RNAshapes -f pharaonis.seq -s -e 5 -m '[[][][]]'
        GGGCCCAUAGCUCAGUGGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUGGGUUCGAAUCCCAGUGGGUCCA
-31.60  (((((((..((((.......)))).(((((((...)))))))).....(((((.......)))))))))))).  [][][]
-31.60  (((((((..((((.......)))).((((((.....)))))).....(((((.......)))))))))))).  [][][]
-31.70  ((((((...((((.......)))).(((((((...)))))))).....(((((.......)))))).))))))  [][][]
-31.70  ((((((...((((.......)))).((((((.....)))))).....(((((.......)))))).))))))  [][][]
```

The "shape match" option is also available in all other analysis modes of RNAshapes.

Output control. RNAshapes offers a number of options to control the program output. The first option, **-S**, splits structures into smaller parts. This is especially useful when

working with long sequences where the program output can be quite confusing for manual inspection. For example:

```
RNAshapes -f pharaonis.seq -S 50
-35.90  1                                              50
        GGGCCCAUAGCUCAGUGGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUG
        ((((((((((((((.((((.....((((((.....)))))).)))))))
        51                  72
        GGUUCGAAUCCCAGUGGGUCCA
        ))))........)))))))).                            []
```

The option **-O** can be used to "fine-tune" the format of the printed results, for example when we are only interested in parts of the result, or when results of RNAshapes should be used as input for other programs. See Section B.2 for a detailed description.

B.2 Options

The following section gives a complete description of the RNAshapes command line interface.

-h Display command option overview

-H `option` Display detailed information on `option`

This displays the corresponding section of the RNAshapes manual for the given command line option.

-v Show version

This shows the version number of RNAshapes.

B.2.1 Sequence analysis modes

-a Shape folding (standard mode)

RNA folding based on abstract shapes. This is the standard mode of operation when no other options are given. It calculates the shapes and the corresponding shreps based on free energy minimization. The energy range can be set by **-e** and **-c**. When not specified, the energy range is set to 10% of the minimum free energy.

-s Complete suboptimal folding

Complete suboptimal folding of RNA. This mode uses a non-ambiguous grammar that also handles dangling bases of multiloop components in a non-ambiguous way. The energy range can be set by **-e** and **-c**. When not specified, the energy range is set to 10% of the minimum free energy.

-p Shape probabilities

Shape probability mode. This option calculates the shape probabilities based on partition function. The probability of a shape is the sum of the probabilities of all structures that fall into this shape. On a computer with 2GB main memory, sequences up to a length of 300 bases can be processed with this mode.

-q Shape probabilities (including shreps)

Shape probability mode. Calculates the shape probabilities based on partition function. This is the same as **-p**, and additionally, the corresponding shreps with their minimum free energies are calculated. Note that this mode is slightly slower than **-p** and can be used with sequences up to a length of 250 bases.

-P `value` Shape probabilities for mfe-best shapes

Shape probability mode. This mode first calculates the best `value` shapes based on free energy minimization. In a second step, it calculates the probability for each of these best shapes. This mode has lower memory requirements than modes **-p** and **-q** and can be used for longer sequences (up to 500 bases). The energy range must be specified with **-e** or **-c** in order to get the desired number of results.

-i `value` Sampling with `value` iterations

Probabilistic sampling based on partition function. This mode combines stochastic sampling with a-posteriori shape abstraction. A sample from the structure space holds M structures together with their shapes, on which classification is performed. The probability of a shape can then be approximated by its frequency in the sample.

Sequences up to a length of around 1500 can be handled with this mode. In our experience, 1000 iterations are sufficient to achieve reasonable results for shapes with high probability.

-C Consensus shapes (RNAcast)

For a family of RNA sequences, this method independently enumerates the near-optimal abstract shape space, and predicts as the consensus an abstract shape common to all sequences. For each sequence, it delivers the thermodynamically best structure which has this common shape. Since the shape space is much smaller than the structure space, and identification of common shapes can be done in linear time (in the number of shapes considered), the method is essentially linear in the number of sequences. Input for RNAcast must be provided in multiple fasta format.

We propose to use the output of the consensus shapes analysis as input for RNAforester [23], a multiple RNA structure alignment program. Use output type **-o** f together with option **-C** to generate suitable input for RNAforester. For example:

```
RNAshapes -f test.fasta -C -o f | RNAforester -m
```

Note that with output type **-o** f only the result for the first consensus is printed (otherwise RNAforester would not work properly). Use the shape match option **-m** to get alternative results. RNAforester is now part of the Vienna RNA package and can be downloaded at

http://www.tbi.univie.ac.at/~ivo/RNA/.

B.2.2 Additional modes (use with any of the above)

-r Calculate structure probabilities

This calculates the probability of every computed structure. It can be combined with any sequence analysis mode. Note that this option increases processing time of modes **-a**, **-s** and **-C**.

-w `value` Specify window size

Beginning with position 1 of the input sequence, the analysis is repeatedly processed on subsequences of the specified size. After each calculation, the results are printed out and the window is moved by the window position increment (**-W**), until the end of the input sequence is reached.

-W `value` Specify window position increment (use with **-w**) (default: 1)

This specifies the increment for the window analysis mode (**-w**).

-m `shape` Match shape (use with **-a**, **-s**, **-p**, **-q**, or **-C**)

Specify a shape for the corresponding mode of operation. For example, with options **-p -m** '[]' the probability of shape [] is computed.

B.2.3 Analysis control

-e `value` Set energy range (kcal/mol)

This sets the energy range for shape folding (**-a**), complete suboptimal folding (**-s**), probability analysis with **-P**, and consensus shapes analysis (**-C**). `value` is the difference to the minimum free energy for the sequence.

-c `value` Set energy range (%) (default: 10)

This sets the energy range as percentage value of the minimum free energy. For example, when **-c** 10 is specified, and the minimum free energy is -10.0 kcal/mol, the energy range is set to -9.0 to -10.0 kcal/mol.

-t `value` Specify shape type (1-5) (default: 5)

The shape type is the level of abstraction or dissimilarity which defines a different shape. In general, helical regions are depicted by a pair of opening and closing square brackets and unpaired regions are represented as a single underscore. The differences of the shape types are due to whether a structural element (bulge loop, internal loop, multiloop, hairpin loop, stacking region and external loop) contributes to the shape representation: Five types are implemented. Their differences are shown in the following example:

```
AUCGGCGCACAGGACAUCCUAGGUACAAGGCCGCCCGUU
..(((.((..(((....))).(((.....)))))))..
```

Type 5: Most abstract - helix nesting pattern and no unpaired regions

```
[[][]]
```

Type 4: helix nesting pattern and unpaired regions in external loop and multiloop

```
_[_[]_[]]_
```

Type 3: nesting pattern for all loop types but no unpaired regions

```
[[[][]]]
```

Type 2: nesting pattern for all loop types and unpaired regions in external loop and multiloop

```
_[[_[]_[]]]_
```

Type 1: Most accurate - all loops and all unpaired

```
_[_[_[_]_[_]]]_
```

-F `value` Set probability cutoff filter (use with **-p**, **-q** or **-P**)

This option sets a barrier for filtering out results with very low probabilities during calculation. The default value here is 0.000001, which gives a significant speedup compared to a disabled filter. Note that this filter can have a slight influence on the overall results. To disable this filter, use option **-F** 0.

-T `value` Set probability output filter (use with **-p**, **-q** or **-P**)

This option sets a filter for omitting low probability results during output. Unlike **-F**, this option does not have any influence on probabilities beyond this value.

-M `value` Set maximum loop length (default: 30) (use -M n for unrestricted)

This option sets the maximum lengths of the considered internal and bulge loops. The default value here is 30. Note that this restriction can have a very slight influence on the calculated structure and shape probabilities. For unrestricted loop lengths, use option **-M** n. This will increase calculation times and memory requirements.

-l Allow lonely base pairs

In default mode, RNAshapes only considers helices of length 2 or longer. With option **-l**, lonely base pairs are also included.

-u Ignore unstable structures (use with **-a**, **-s** or **-C**)

This option filters out closed structures with positive free energy.

B.2.4 Input/Output

-o `value` Specify output type (1-4,f) (default: 2)

Specifies the output type. Output type 1 mimics RNAfold and RNAsubopt. Type 2 is the default RNAshapes output. Type 3 is similar to type 2, but without parentheses and with only a single space between results. This output type can be used for exporting results as a comma separated text-file to other applications like Microsoft Excel. Type 4 is a colored variant of type 2. Additional output types can be defined with option **-O**.

In consensus shapes analysis (**-C**), output type f can be used to generate suitable input for RNAforester (a multiple RNA structure alignment program; see **-C** for details).

-O `string` Specify output format string

The option **-O** can be used to "fine-tune" the format of the printed results, for example when we are only interested in parts of the result, or when results of RNAshapes should be used as input for other programs. The syntax is as follows:

```
TYPE{FORMAT}...TYPE{FORMAT}
```

where TYPE specifies the result element:

```
D: structure in dot-bracket notation
S: shape string
E: energy
P: shape probability
R: structure probability (option -r)
C: shape rank (option -C)
V: verbatim output, independent of result element
```

FORMAT is the C-format string that shall be used to print the corresponding result element. Typical C-format strings are %.2f for a floating point number with two decimal places and %s for a string. For example, to print only the structure followed by its energy, we can use -O 'D{%s\t}E{%.2f}V{\n}'. The symbol '\n' performs a line break, the symbol '\t' a tabulator. An ANSI escape sequence can be used with symbol '\e' (see Example 4 below).

The standard output types (option **-o**) are defined as follows:

```
1) 'D{%s  }E{(%.2f)  }R{(%.7f)  }P{%.7f  }S{%s}C{  R = %d}V{\n}'
2) 'E{%-8.2f}R{(%.7f)  }D{%s  }P{%.7f  }S{%s}C{  R = %d}V{\n}'
3) 'E{%.2f }R{%.7f }D{%s }P{%.7f }S{%s}C{ %d}V{\n}'
4) 'E{%-8.2f}R{(%.7f)  }D{\e[1;31m%s\e[0m  }
    P{\e[1;30m%.7f\e[0m  }S{%s}C{  R = %d}V{\n}'
```

-S `value` Specify output width for structures

This splits the structure strings into parts of the specified length. This option is useful when displaying results for long sequences that would otherwise not fit onto the screen.

-# `value` Print only the first `value` results

This option specifies the total number of results to be printed. When this number is reached, the program terminates. Note that this option does not reduce calculation time or memory requirements (except for modes **-s** and **-i**).

-g `value` Generate structure graphs for first `value` structures

This generates postscript structure graphs for the first `value` structures computed for a sequence. If multiple sequences are given, `value` graphs are generated for each sequence.

The filenames of the structure graphs consist of several parts:

1. When the input sequence is given in fasta format, the first 12 characters of the sequence description are taken. White-spaces and special characters are removed. When no description is available, "rna" is chosen as standard name.

2. The sequence position in window mode (option **-w**).

3. The running number of the result.

For example, the first result of a sequence called "xyz" at position 7 in window mode will be saved in file xyz_7_1.ps.

-L Highlight uppercase characters in structure graphs

Used with option **-g**, this generates postscript structure graphs where all uppercase characters in the input sequence are highlighted. This option is useful for marking interesting regions of the input sequence.

-N Do not include additional information in graph output file

In standard operation, the postscript structure graph generation (option **-g**) generates files with shape, energy, and shape probability (if available) included at the bottom. Use this option to suppress this.

-f `file` Read input from `file`

Let RNAshapes load its input data from `file`. `file` can contain a plain single sequence, or multiple sequences in fasta format. When given multiple sequences, each sequence is processed separately in the order of input.

Valid characters in an input sequence are "ACGU" and "acgu". "T" and "t" will be converted to "U". Other letters are mapped to "N" and will not be paired. All other characters are ignored.

-B Show progress bar (use with **-p**, **-q** or **-P**)

Setting this option activates a progress bar. This is useful when experimenting with options **-p** and **-q**, to get an impression of the expected running time.

-z Enable colors (in interactive mode: disable colors)

This option enables colored output. In interactive mode, this is the default setting, so use **-z** to disable colors here.

-Z Enable colors for dotbracket and shape strings

This option colors dotbracket and shape strings in the result output, such that corresponding structural elements have the same color in both representations.

-D `string` Convert dotbracket-string to shape (choose type with **-t**)

Convert a dotbracket-string into a shape. Choose the shape type with option **-t**. The default shape type is 5. For example:

```
RNAshapes -D '.((((.(((....))).))))...(((....))).' -t 4
_[]_[]_
```

-U Start graphical user interface

This option starts the graphical user interface included in the RNAshapes distribution. It requires Java 1.4.2 or later (download from http://java.sun.com/). Note that the RNAshapes distribution for Microsoft Windows includes a slightly different user interface. It does not require Java and additionally, it offers an interactive visualization of the calculated RNA structures.

B.2.5 Additional interactive mode commands

:s Show current configuration

This command shows the current settings in an interactive session.

:d Reset configuration

This command sets all settings to their default values.

:e `string` Execute system command

Command **:e** executes a system command. For example, we can use the command **:e** gv rna_1.ps to open a structure graph file created with option **-g** (on a unix machine with gv installed).

:q Quit
This command quits an interactive RNAshapes session.

Bibliography

[1] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation and Compiling.* Prentice-Hall, Englewood Cliffs, NJ, 1973. I and II.

[2] R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[3] G. L. Burn, C. L. Hankin, and S. Abramsky. The theory and practice of strictness analysis for higher order functions. In *LNCS 217.* Springer-Verlag, New York, NY, 1985.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, New York, NY, 1977. ACM.

[5] D. M. Cvetković, M. Doob, and H. Sachs. *Spectra of Graphs - Theory and Applications.* Academic Press, New York, NY, 1980.

[6] Y. Ding and C. E. Lawrence. A statistical sampling algorithm for RNA secondary structure prediction. *Nucleic Acids Res.*, 31:7280–7301, 2003.

[7] R.D. Dowell and S.R. Eddy. Design of lightweight stochastic context-free grammars for RNA secondary structure prediction. *BMC Bioinformatics*, 5(71), 2004.

[8] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis.* Cambridge University Press, 1998.

[9] D. Evers and R. Giegerich. Reducing the conformation space in RNA structure prediction. In *German Conference on Bioinformatics*, 2001.

[10] F. R. Gantmacher. *The Theory of Matrices*, volume one. Chelsea Publ. Comp., New York, NY, 1959.

[11] P.P. Gardner and R. Giegerich. A comprehensive comparison of comparative RNA structure prediction approaches. *BMC Bioinformatics*, 5(140), 2004.

[12] W. Gautschi. The asymptotic behaviour of powers of matrices. *Duke Math. J.*, 20:127–140, 1953.

[13] R. Giegerich. A declarative approach to the development of dynamic programming algorithms, applied to RNA folding. Technical report, Universität Bielefeld, 1998.

[14] R. Giegerich. Explaining and controlling ambiguity in dynamic programming. In *Proc. Combinatorial Pattern Matching*, pages 46–59. Springer LNCS 1848, 2000.

[15] R. Giegerich. A systematic approach to dynamic programming in bioinformatics. *Bioinformatics*, 16:665–677, 2000.

[16] R. Giegerich and C. Meyer. Algebraic Dynamic Programming. In Hélène Kirchner and Christophe Ringeissen, editors, *Algebraic Methodology And Software Technology, 9th International Conference, AMAST 2002*, pages 349–364. Springer LNCS 2422, 2002.

[17] R. Giegerich, C. Meyer, and P. Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, 2004.

[18] R. Giegerich and P. Steffen. Implementing algebraic dynamic programming in the functional and the imperative programming paradigm. In E.A. Boiten and B. Möller, editors, *Mathematics of Program Construction*, pages 1–20. Springer LNCS 2386, 2002.

[19] R. Giegerich and P. Steffen. Challenges in the compilation of a domain specific language for dynamic programming. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, 2006.

[20] R. Giegerich, B. Voß, and M. Rehmsmeier. Abstract Shapes of RNA. *NAR*, 32(16):4843–4851, 2004.

[21] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

[22] J.H. Havgaard, R.B. Lyngsø, G.D. Stormo, and J. Gorodkin. Pairwise local structural alignment of RNA sequences with sequence similarity less than 40%. *Bioinformatics*, 21(9):1815–1824, 2005.

[23] M. Höchsmann, B. Voss, and R. Giegerich. Pure multiple RNA secondary structure alignments: A progressive profile approach. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1(1):53–62, 2004.

[24] I.L. Hofacker, W. Fontana, P.F. Stadler, L.S. Bonhoeffer, M. Tacker, and P. Schuster. Fast folding and comparison of RNA secondary structures. *Monatshefte f. Chemie*, 125:167–188, 1994.

[25] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.

[26] G. Hutton. Higher order functions for parsing. *Journal of Functional Programming*, 3(2):323–343, 1992.

[27] S. Jain and S. Krishna. Emergence and growth of complex networks in adaptive systems. *Comp. Phys. Comm.*, 121-122:116–121, 1999.

[28] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[29] J. M. Lewis and M. Yannakakis. The node-deletion problem for hereditary properties is NP-complete. *Journal of Computer and System Sciences*, 20(2):219–230, April 1980.

[30] D.H. Mathews and D.H. Turner. Dynalign: an algorithm for finding the secondary structure common to two RNA sequences. *J Mol Biol*, 317(2):191–203, 2002.

[31] J.S. McCaskill. The equilibrium partition function and base pair binding probabilities for RNA secondary structure. *Biopolymers*, 29:1105–1119, 1990.

[32] U. Möncke and R. Wilhelm. Grammar flow analysis. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, pages 151–186. LNCS 545, 1991.

[33] T. L. Morin. Monotonicity and the principle of optimality. *Journal of Mathematical Analysis and Applications*, 86:665–674, 1982.

[34] R. Nussinov, G. Pieczenik, J.R. Griggs, and D.J. Kleitman. Algorithms for loop matchings. *SIAM J. Appl. Math.*, 35:68–82, 1978.

[35] L. Pachter and B. Sturmfels, editors. *Algebraic Statistics for Computational Biology*. Cambridge University Press, 2005.

[36] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Series in Computer Science. Prentice-Hall International, 1987.

[37] S.L. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, April 2003.

[38] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. International Computer Science Series. Addison-Wesley, 1993.

[39] J. Reeder and R. Giegerich. Design, implementation and evaluation of a practical pseudoknot folding algorithm based on thermodynamics. *BMC Bioinformatics*, 5(104), 2004.

[40] J. Reeder and R. Giegerich. Consensus shapes: an alternative to the Sankoff algorithm for RNA consensus structure prediction. *Bioinformatics*, 21(17):3516–3523, 2005.

[41] J. Reeder, P. Steffen, and R. Giegerich. Effective ambiguity checking in biosequence analysis. *BMC Bioinformatics*, 6(153), 2005.

[42] M. Rehmsmeier, P. Steffen, M. Höchsmann, and R. Giegerich. Fast and effective prediction of microRNA/target duplexes. *RNA*, 10:1507–1517, 2004.

[43] G. Rote. Path problems in graphs. In G. Tinhofer, E. Mayr, H. Noltemeier, and M. Syslo, editors, *Computational Graph Theory*, Computing Supplementum 7, pages 155–189. Springer-Verlag, 1990.

[44] D. Sankoff. Simultaneous solution of the RNA folding, alignment and protosequence problems. *SIAM J. Appl Math*, 45(5):810–825, 1985.

[45] A. Sczyrba, J. Krüger, H. Mersch, S. Kurtz, and R. Giegerich. RNA-related tools on the Bielefeld Bioinformatics Server. *Nucl. Acids. Res.*, 31(13):3767–3770, 2003.

[46] D.B. Searls. Linguistic approaches to biological sequences. *CABIOS*, 13(4):333–344, 1997.

[47] B.A. Shapiro. An algorithm for comparing multiple RNA secondary structures. *CABIOS*, 4:381–393, 1988.

[48] P. Steffen. Basisfunktionen für die Übersetzung von Programmen der Algebraischen Dynamischen Programmierung. Diplomarbeit, Universität Bielefeld, 2002.

[49] P. Steffen and R. Giegerich. Versatile and declarative dynamic programming using pair algebras. *BMC Bioinformatics*, 6(224), 2005.

[50] P. Steffen and R. Giegerich. Table design in dynamic programming. *Information and Computation*, 204(9):1325–1345, 2006.

[51] P. Steffen, B. Voß, M. Rehmsmeier, J. Reeder, and R. Giegerich. RNAshapes: an integrated RNA analysis package based on abstract shapes. *Bioinformatics*, 22(4):500–503, 2006.

[52] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[53] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[54] R. S. Varga. *Matrix Iterative Analysis*. Prentice-Hall, 1962.

[55] B. Voß, R. Giegerich, and M. Rehmsmeier. Complete probabilistic analysis of RNA shapes. *BMC Biology*, 4(5), 2006.

[56] R. Wilhelm and D. Maurer. *Übersetzerbau: Theorie, Konstruktion, Generierung*. Springer-Verlag, Berlin, 2 edition, 1996.

[57] S. Wuchty, W. Fontana, I. L. Hofacker, and P. Schuster. Complete suboptimal folding of RNA and the stability of secondary structures. *Biopolymers*, 49:145–165, 1999.